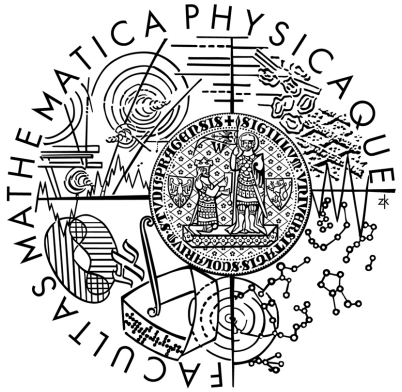
Charles University, Prague, Czech Republic
Faculty of Mathematics and Physics

MASTER THESIS



Jaroslav Urban

Deployment Planner for Heterogeneous Component-based Applications

Department of Software Engineering
Supervisor: Ing. Lubomír Bulej Ph.D.
Study program: Computer Science, Software Systems

I would like to thank my advisor Lubomír Bulej for his excellent support and guidance during the course of this work. His honest interest in the project, assistance and expert knowledge in the problem field were of great help. I would also like to thank Pavel Šafrata for his help with the integration with his deployment runtime.

Finally I thank my family and friends for their strong support during my work on the thesis.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on 28^{th} July 2008 Jaroslav Urban

Contents

Abstract

Název práce: Deployment Planner for Heterogeneous Component-based Applications

Autor: Jaroslav Urban

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Ing. Lubomír Bulej Ph.D.

e-mail vedoucího: bulej@nenya.ms.mff.cuni.cz

Abstrakt:

Nasazování komponentových aplikací je proces, který je zpřístupňuje uživatelům k dalšímu používání. OMG Deployment & Configuration Specification je dokument, jehož cílem je vytvoření jednotného nasazovacího řešení nezávislého na komponentových modelech.

OMG specifikaci používáme k vytvoření prostředí pro nasazování heterogenních komponentových aplikací, které jsou implementovány pomocí více komponentových modelů. Tato práce je zaměřena na plánovací fázi specifikace, která vybírá počítačové zdroje pro komponentové aplikace s ohledem na jejich požadavky. Plánovací fázi jsme implementovali pomocí grafického uživatelského rozhraní využívajícího automatický plánovací algoritmus, který pomáhá uživateli s vytvořením platného naplánování. Plánovací nástroje jsou rozšiřitelné o podporu pro další technologie, komponentové modely a plánovací heuristiky.

Klíčová slova: *komponentové aplikace, heterogenní komponentové aplikace, plánování*

Title: Deployment Planner for Heterogeneous Component-based Applications

Author: Jaroslav Urban

Department: Department of Software Engineering

Supervisor: Ing. Lubomír Bulej Ph.D.

Supervisor's email address: bulej@nenya.ms.mff.cuni.cz

Abstract:

Deployment of component applications is the process of making them available for further use by clients. The OMG Deployment & Configuration Specification aims at creating a unified deployment process independent of component models.

We use the OMG specification to create a deployment framework able to deploy heterogeneous component applications which are implemented using multiple component models. This work focuses on the planning phase of the specification, which selects computer resources for components with respect to their requirements. We have implemented the planning phase via a graphical user interface utilizing an automated planning algorithm which assists the user in creating a valid planning. The planning tools are extensible to support additional technologies, component models and more advanced planning heuristics.

Keywords: *component applications, heterogeneous component applications, planning*

1 Introduction

1.1 Component Applications and Programming

Software systems can be designed and implemented using various approaches and techniques. An important problem which many try to solve is modularity. Importance of modularity rises as software system sizes grow. Software modularity has multiple benefits, as the ability to easily integrate software produced by third parties to reduce development time, better fault isolation and easier maintenance and future development.

One approach to modularity is *component based programming*. This programming paradigm uses the abstraction of a *software component* as its principal entity. Software components are a form in which software can be described, distributed, launched and used. A software component describes its behavior and requirements with contractual information about its interface. A software component is also a unit of deployment which contains the software code and can be deployed transparently in different contexts.

Software components are reusable in different contexts without the need to modify their internals. Third-party users or developers can deploy and re-use the same component without the need to understand its internals.

Complex applications can be built in a modular way by composing and inter-connecting multiple components, thus software components are also a unit of composition. To achieve the functionality of complex multi-component systems, the cooperating components communicate with each other. The communication can span networks, and thus the components can be used to construct distributed systems.

A useful mechanism for creating complex components is composing them from smaller components, which can be recursively created from other components. Such hierarchical components delegate parts of their responsibilities (i.e. interfaces) onto their child components, and interconnect the child components between themselves.

Component applications must be packaged in a pre-defined way and contain such metadata so that they are understandable to other systems, frameworks and tools. A conceptual model of components which describes their structure, used abstractions and their semantics is a *component model*. Numerous component models exist, for example *EJB* [1], *CORBA Component Model* [2] , *SOFA* [4] [5] and *Fractal* [3].

1.2 OMG D&C Specification

Before a component application can be used, it must be deployed in a deployment framework. The framework is a software system which can instantiate the components, manage their installation, lifecycle and configuration.

Currently most component models have their own specific deployment frameworks which are incompatible with others and support only the one component model. Some component models (EJB) even have several incompatible deployment frameworks.

On the other hand, most frameworks solve very similar problems in a similar way. Their basic concepts and workflows are quite similar, and most of their differences are in the realm of incompatible data models.

To enhance the cooperation between multiple component models and to unify the deployment process, the Object Management Group (OMG) created the Deployment and Configuration of Component-based Distributed Applications Specification [6]. It describes the deployment framework from both the workflow view and the data model view.

The workflow as described by OMG D&C has several parts. *Installation* is the act of taking published and packaged component applications and making them available in the deployment framework. Metadata of the components which describe their structure, requirements etc are stored in a *component repository*. This step does not include copying the component binaries to the target environment where they will be instantiated.

Installed components can be *configured*. This allows the user to control the runtime behavior of running components (e.g. maximum number of threads, caching policy). Component configurations can be persisted and used again later.

Before the components are instantiated and ready to use, decisions must be made about where to instantiate them and how to interconnect them. Computer hosts must be selected for all the components in such a way which doesn't break the requirements of the components or the structure of the whole component application. These decisions are described in a *deployment plan* , which is created in the *planning* phase.

Instantiation and lifecycle management of components is handled by the *runtime* part of the deployment framework. The runtime uses the information created in the planning phase to transport the component data to the target environment (i.e. specific computer hosts) and create their instances. The runtime provides mechanisms to manage and monitor the running component instances.

The OMG D&C specification describes the business interfaces which implement the workflow and the data model used throughout the deployment process in its various parts. The data model is independent of component models, and support for specific component models is added through enhancement of the base platform independent model.

Usually when complex software is created by composition of multiple smaller components, it is required that all of the components are using the same component model, programming language etc. A significant step forward is the ability to mix components from different component models to

assemble a *heterogeneous component application*. A deployment framework for heterogeneous applications must be independent of the specific implementations of the components, and it must have a data model which can describe the components' interfaces and requirements independently of their component models. We are using the OMG D&C specification to create a framework capable of deploying heterogeneous component applications constructed from arbitrary components. We do not modify the data model of the specification to the needs of the specific component models, but we use the original model to store all information required by the component models.

1.3 Planning

As outlined in the previous section, the workflow described by the OMG D&C specification contains a planning phase of the component application deployment workflow. From the point of view of the user who wants to execute a component application, this is the phase where he must make the decisions of where the component instances will be created. It is a highly interactive part of the deployment framework, which requires a user friendly graphical user interface.

Planning decisions require cooperation with other parts of the deployment framework. The other parts provide the information required for creating a valid deployment plan. The OMG D&C specification describes repositories for storing the structure and requirements of the component applications and description of the target environment topology and available resources. This information needs to be presented to the computer user in a clear way, so they can be used for his planning decisions.

Planning of a complex component applications is a very difficult task. It involves planning decisions for a high number of components, potentially counting in hundreds. Components have various dependencies and requirements, which must be evaluated and resolved correctly so that the resulting deployment plan is valid. The requirements are of various types with different semantics, thus further complicating the planning.

1.4 Goals

Because of the complexity of the planning, we believe that automated or semi-automated planning tools are needed. The tools should achieve several goals:

- **Automated planning:** The planning tools will use an algorithm which will assist the user in creating a valid deployment plan. The algorithm will try to automatically or semi-automatically find valid plannings based on user input.
- **Graphical user interface:** The user interactions needed for creating a deployment plan are highly interactive and require a graphical user interface. The user interface must assist the user in creating the plan with the help of the automated planning tools. The user interface must also provide specific views of all the information needed for planning, like the description of the target environment.
- **Heterogeneous component applications:** The planning tools must be able to plan heterogeneous component applications which use multiple component models for its components.

1.5 Structure of this Work

This work starts with a description and analysis of the OMG D&C specification in section 2. We explore the various models defined in the specification thus providing a basic framework for the design of the planning tools.

The planning tools must be able to plan heterogeneous component applications. The difference between our view of heterogeneity and the OMG's view is explained in section 3. This section also analyses the heterogeneity related requirements of other parts of our deployment framework.

The previous sections allow us to analyze the problem being solved more closely. We revisit the goals of our work and redefine them in more detail in section 4.

The implementation of the planning tools is separated into two basic parts, a planner and a planning GUI. The overall design of the planner is described in section 5 and its implementation is elaborated in section 6. The planning GUI is described in section 7.

Summary of the basic properties of the implementation as well as evaluation of the work with respect to the goals of the thesis is presented in section 8. The OMG D&C specification is reviewed in the evaluation too. This work is a part of an ongoing research project and is designed to be extensible. Description of future work in terms of integration with other parts of our deployment framework and in terms of possible future enhancements is given in section 8 together with an evaluation with respect to related work.

Section 9 concludes this work and provides a brief summary. It is followed by a description of the contents of the DVD distributed with this work in section 10. The DVD contains the implementation of the planning tools, source code and also this text in electronic form.

2 OMG Deployment & Configuration Specification

2.1 Overview

The OMG Deployment & Configuration specification [6] describes the workflow and data model for a component application deployment framework. The goal of the specification is to provide a common model for component application deployment frameworks.

The specification supports deployment of complex distributed systems. It assumes that there is a target environment consisting of multiple interconnected computer nodes which together constitute a domain. The nodes are connected to networks, and the domain can contain multiple networks. Communication spanning multiple networks is possible via bridges which connect the networks.

The specification embraces the concept of hierarchical components by supporting components which are either simple monolithic components (without child components) or complex assemblies of subcomponents. Component assemblies define the connections between their subcomponents and connections which delegate the assembly's interface responsibilities.

The component metadata in the specification describe not only the logical structure of the component applications, but also the requirements and dependencies of the components. This way the component metadata encapsulate all information required to successfully create a component instance on a computer node with a valid environment.

The data model and business interfaces of the specification handle all interactions used in the deployment of component applications:

- **Installation:** Components must be installed in the framework before they can be used. The installation process stores a complete description of the component in a repository (i.e. the component metadata), from where it is available to the other parts of the deployment framework.
- **Configuration:** Components usually need to be configured before they can be instantiated. The configuration of the components is stored in a repository, so it can be reused later.
- **Planning:** Prior to instantiation of the components, decisions must be made about where will the instances be created. The resulting planning decisions must consider component dependencies, resource requirements of the components with respect to the resources available in the target domain etc.
- **Preparation and launch:** When the planning decisions are complete, component instances can be created on the selected nodes. Before the

instantiation, the component code must be made available in the target computer node and the environment of the node must be prepared. This phase is handled by the *deployment runtime*.

Deployment workflow is described via three models in the specification: component model, target model and execution model. The component model handles the component application metadata. The target model describes the computer domain available for deployment. Launch and lifecycle management of the component applications is handled by the execution model. All three models have two parts, a data model and a management model. The data model is used for communication of the various parts of the deployment framework. The management model describes the business interfaces which implement the functionality required by the deployment workflow.

2.2 Component Data Model

Overview

The component data model describes the logical structure of components, their requirements and dependencies, and the locations of the component binary data. This model contains all information about components which is necessary to make the planning decisions. Parts of the deployment framework that handle the launching and instantiation of components do not use this model, but a simpler execution model described in section 2.6 .

The component data model embraces the concept of hierarchical components. Such components are created by the composition of multiple subcomponents. The subcomponents can be recursively composed of other components. Communication between components is handled by connections. A component can either have a concrete (*monolithic*) implementation consisting of one or more *artifacts* (library files, executables), or it is an *assembly* of multiple sub-components.

Components describe their behavior contractually by defining their interface. A component interface specifies *ports* of the component, which represent services provided or required by the component. The interface also specifies configuration properties of the component which can modify its runtime behavior.

Requirements and dependencies of the components are stored in the component data model. The requirements are of multiple types, for example requirements on hardware resources on the target node. The requirements are not pre-defined by the specification, but arbitrary requirements and resource types can be defined by the implementations of the deployment runtime.

The component data model represents the configuration of components. A specific configuration of a component can be represented as a description of

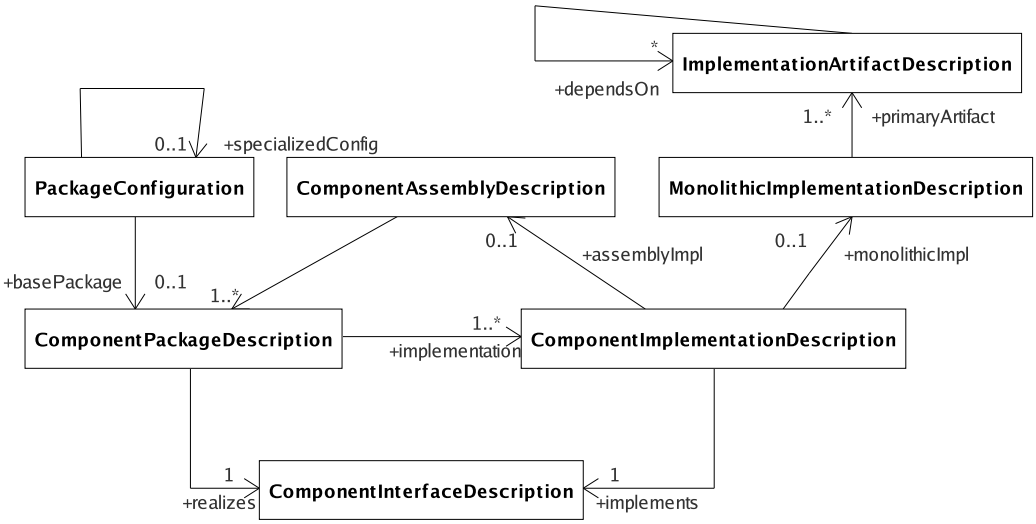


Figure 1: Component data model

new component which specializes the original component. The specialization is done in terms of setting configuration options, or properties.

Heterogeneous component applications in the context of the specification are applications in which components can have multiple implementations. Each implementation can be specific for some platform or environment (e.g. an operating system). Specific implementations must be chosen before the component is instantiated. This concept of heterogeneity is different from ours, in which heterogeneous component applications can contain components from different component models. The latter interpretation of the term is consistently used throughout this work.

Storage and management of the component data model is handled by a component repository. The business interface of the repository is described in the component management model in section 2.3.

Class Overview

The overview of the most important classes of the component data model is provided in figure 1.

A `ComponentInterfaceDescription` describes the interface of the component. The interface describe the services provided and required by the component with the abstraction of a *port*. Port is an endpoint of connections which interconnect components and serve their communication needs. Ports can either provide services (they are the target of connections) or they require a connection to some other service (i.e. port of another component). Ports can be mandatory or optional, and they can impose additional restrictions on the connections which are connected to them. Ports also specify their supported data type. Component interfaces define configuration properties

supported by the component implementation.

A `PackageConfiguration` describes a configured component package, and represents a reusable work product. A `PackageConfiguration` has an optional human-readable label and an optional UUID. This is the top-level element of a component stored in the component repository.

`PackageConfiguration` is a subclass of `ComponentUsageDescription` which describes the re-use of existing packaged components. Existing component packages can be reused in several ways:

- **Base package:** A packaged component can be directly referenced as a value. The packaged component consists of (potentially) multiple implementations, which can have different structure or different requirements on the target platform (e.g. operating system). During the planning phase, an implementation must be chosen which meets the requirements of the `ComponentUsageDescription`.
- **Configuration:** Components can be configured before their launch, and the configuration can be persisted. The configuration is represented by a component package which references an existing (unconfigured) package but also specifies the values of some configuration properties.
- **Import:** A component package stored in a location outside the repository can be referenced. During the installation of such component metadata into the repository, the package import is resolved and the contents of the referenced package is copied into the repository.
- **Repository reference:** An existing component package in the repository is referenced by its installation name, UUID or component interface.

A component package defined by a `ComponentPackageDescription` must have a defined component interface and contain at least one component implementation. Each component implementation can have implementation capabilities defined, which can be used during the planning to select a specific implementation for a component. A component implementation can be either a simple concrete (*monolithic*) implementation (with no subcomponents) or it can be an *assembly* of multiple subcomponents.

Monolithic Implementation

A monolithic implementation contains descriptions and locations of binary artifacts (e.g. libraries or executable files) which are required to create the instance of the component. The locations of the artifacts are used by the runtime to transport the artifact data to the target node automatically. Artifact data is not necessarily stored in the component repository, but it must

be available from a location supported by the deployment runtime (e.g. an HTTP URL). Artifacts can have dependencies on other artifacts, which must be resolved by the deployment runtime. The monolithic implementation also describes parameters or arguments passed onto the deployment runtime which are used to instantiate the component.

Monolithic implementations and their artifacts impose requirements on the computer nodes where they are instantiated. The requirements of the components can be requirements on the hardware configuration, on installed software or in general on capabilities of the target environment. The OMG D&C specification does not define specific requirements and resources such as memory size, CPU speed etc. The specification allows us to define custom resources and requirements, and describe their semantics. For example, we can specify a `memory` resource on the target node. The resource has a `size` property with a specific value, which is of the `capacity` type. Capacity type means that this property is consumed by its users, so that the value of the property is decreased each time it is allocated by a user. The specification describes multiple types of properties, which cover a wide range of possible resource types. Such description of resources contains the semantics necessary to use the resource correctly and reference it in requirements. The requirements can also use existing component instances as required resources, or they can define that the monolithic implementation serves as a resource.

Assembly Implementation

Hierarchical components are described via component assemblies modeled by a `ComponentAssemblyDescription`. An overview of the assembly class model is provided in figure 2. An assembly implementation of a component describes the subcomponents which are composed to form the component assembly. The subcomponents are interconnected so they can communicate, and some responsibilities and requirements of the component assembly are delegated to the subcomponents.

The subcomponents of the assembly are associations with a subclass of a `ComponentUsageDescription` (which is omitted from the provided figures for clarity). Each subcomponent has a name, which is unique in the context of the component assembly. The assembly can define `Locality` constraints on its subcomponents. The locality constraints specify requirements on the relative positions of the subcomponent instances in the target environment or the process separation of the component instances. For example, a locality constraint can demand that two heavily communicating subcomponents are instantiated on the same node but in different processes for performance reasons. Another example is two hardware demanding subcomponents that have to be planned onto different nodes. The specification defines several types of locality constraints.

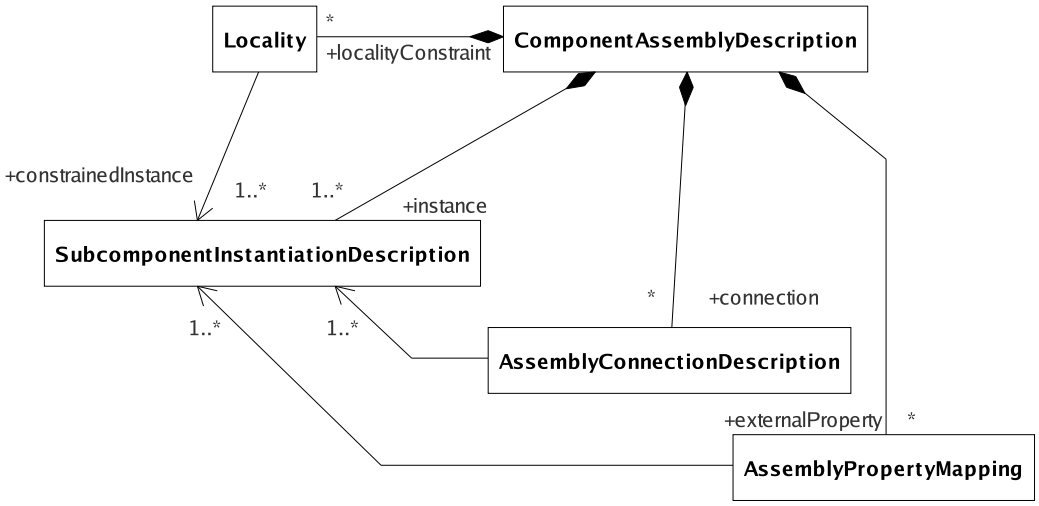


Figure 2: Component assembly

An `AssemblyConnectionDescription` represents a connection between the ports of multiple subcomponents. A connection can also connect outside ports of the component assembly interface to ports of the subcomponents. This way the assembly usually delegates most of its responsibilities onto its subcomponents or represents the requirements of the subcomponents as its own. All non-local connections must be planned onto a physical network, which must meet the connection's requirements. Connection requirements are defined similarly to component requirements, i.e. they are not pre-defined by the specification.

Component interfaces describe the configuration parameters of components. Component assemblies can delegate their configuration parameters onto subcomponents via an `AssemblyPropertyMapping`.

A `ComponentAssemblyDescription` is considered to be purely virtual, i.e. no instance of assembly itself is created at runtime. Consequently, assembly does not have any artifacts to instantiate it. The OMG D&C specification supports only purely virtual component assemblies. Such component assemblies only describe the logical structure of the component application, but they are not instantiated in runtime. Because the assemblies are not instantiable, they do not specify their requirements on the target node. This is a significant difference from various component models as Fractal or SOFA, and we will need to address it in our deployment framework. The OMG D&C specification supports only virtual assemblies because the component model created by OMG, the CORBA Component Model, supports only such assemblies.

2.3 Component Management Model

The component management model describes the interface of a component repository. The repository stores component metadata using the component data model.

The component repository is represented by a `RepositoryManager` class. The repository stores `PackageConfiguration` elements which represent component metadata. Installation of a component is accomplished by installing a `PackageConfiguration` into the repository under a specified name. The name must be unique within the repository. During the installation, a `PackageConfiguration` can either be provided directly as a value, or as a location outside the repository from which it will be imported. Component metadata can be updated by reinstalling a component under the same name. The repository of course provides an interface for the uninstallation of components. The repository can list the names of all installed components and list all supported component types implemented by the installed components. The `PackageConfiguration` elements can contain a packaged component either directly by value in a `ComponentPackageDescription`, or reference one installed in the same repository or in a location outside the repository.

2.4 Target Data Model

The target data model represents information about the target `Domain` which consists of computer nodes and networks. The target data model describes the logical structure of the target domain and the properties of its elements. An overview of the model is provided in figure 3.

The `Node` class represents a physical computer host. A node can be connected to multiple networks. A description of the resources available on the node is available. The resources are used by component instances.

The `Interconnect` class represents a physical network, e.g. a LAN. Multiple nodes can be connected to the interconnect. A description of the resources available on the interconnect is available. These resource are used by connections which connect instances of running components.

The `Bridge` class represents a bridge between multiple physical networks, e.g. a router. Bridges provide indirect communication between nodes which are connected to different interconnects. Bridges can also have resources specified, which are used by the component connections. Bridges significantly complicate the planning process, thus this work will not support them.

Description of resources provided by the elements of the domain is represented by the `Resource` class. Resources can be specific to one element of the domain, or they can be shared by several elements of the domain as a `SharedResource`. The OMG D&C specification does not pre-define the

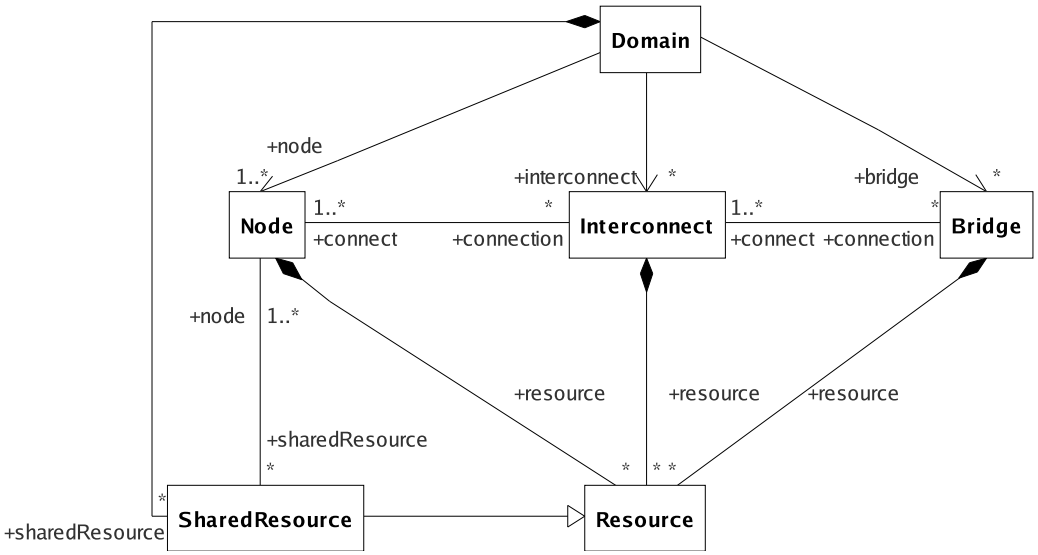


Figure 3: Target data model

resources (e.g. `memory size`), but it provides mechanisms for defining arbitrary resources by the implementers of the specification and by the users. A resource has a specific name (e.g. `memory`) and several properties. Each property has its own name (e.g. `size`), value (e.g. `1024`) and a description of its semantics. The description of its semantics specifies the behavior of the resource and how it satisfies requirements of components. There are several ways defined by the specification of how the resource satisfies a requirement, e.g. a `capacity` resource has a maximum value which is decreased each time it satisfies the requirement of a component.

2.5 Target Management Model

The target management model describes the `TargetManager` interface for manipulating the information about a target domain. Clients can retrieve and update the domain model. The domain model can contain the description of all resources, or just of the available resources. Resources can be allocated by clients to satisfy the requirements of component instances. The resource allocation can be destroyed to free unused resources.

2.6 Execution Data Model

The execution data model represents a prescription called `DeploymentPlan` which is used by the deployment runtime to create the instances of components which compose the whole component application. The deployment plan is the result of the planning phase, so it is the principal output of the

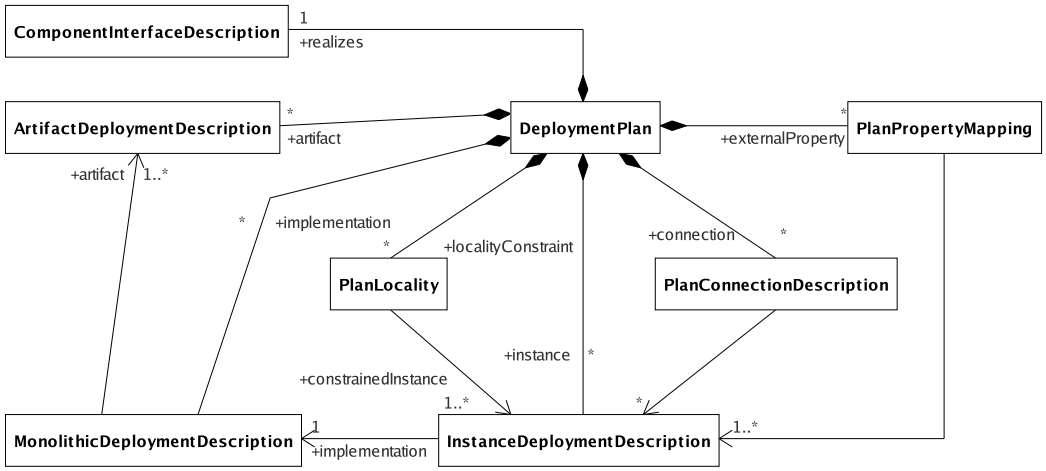


Figure 4: Execution data model

planning tools. An overview of the execution data model is provided in figure 4.

The component application described by the deployment plan implements an interface as defined in a `ComponentInterfaceDescription`. Configuration properties of the component application which are defined in its interface can be propagated onto specific component instances via a mapping which is defined by `PlanPropertyMapping`.

A `MonolithicDeploymentDescription` contains the information needed to create specific component instances. It stores execution parameters which are used by the deployment runtime to create the component instance. Before the component instance is created, its binary artifacts (e.g. libraries, executable code) must be made available on the computer node. Description of the artifacts is stored in `ArtifactDeploymentDescription`. The artifact description contains the location of the artifact data. The location is component repository independent, so it can be retrieved from any location supported by the deployment runtime (e.g. an HTTP server). The artifact description can also contain execution parameters which are used by the deployment runtime, e.g. command line arguments.

The node chosen for a specific component instance during planning is stored in the `InstanceDeploymentDescription`. The instance deployment description also contains component configuration as defined in its interface and a description of the allocation of resources which satisfy the component's requirements.

The deployment plan contains locality constraints which describe the process separation of component instances. A `PlanLocality` can specify that several components must be instantiated in the same process (i.e. address space) or in separate processes.

Component instances are interconnected by the deployment runtime as

specified in `PlanConnectionDescription`. The plan connection description connects ports of component instances. It can also connect the component instances to the external ports of the whole component application which are defined in its interface. The connections also carry a description of the resources allocated to satisfy their requirements.

The execution data model describes an instance of a "flat" component application, which doesn't have any notion of component hierarchy. This is a significant difference from the component data model. On the other hand, the component assemblies in the component data model are not instantiable (see section 2.2), so they cannot be represented in the execution data model.

2.7 Execution Management Model

The `ExecutionManager` interface of the deployment runtime is described in the execution management model. The runtime can create the instances of component applications and manage their lifecycle. A component application instance is created based on the information provided in a `DeploymentPlan`. The specific details of the whole execution management model are not important in the context of this work, as it focuses on the creation of a valid deployment plan which is the principal input of the deployment runtime.

3 Heterogeneous Component Applications

The OMG D&C specification addresses heterogeneous component applications in a different way than we do. The component model of the specification supports components with multiple implementations so they can be deployed in heterogeneous computer systems (e.g. with multiple operating systems). The implementations are often platform dependent and each of them can have a different component structure or it can have different artifacts specific to some platform (e.g. operating system).

In this work we will use the term heterogeneous component applications for applications which can be composed from components implemented in multiple component models as described in section 1.2.

3.1 Component Models

3.1.1 Fractal

The Fractal [3] component model supports components which are composed from multiple subcomponents and are instantiable in runtime. Components can represent resources by being shared between multiple composite components. The Fractal components can be monitored and reconfigured in runtime.

One of the goals of the Fractal component model is the ability to run on a wide range of computer systems, from embedded to mainframes. Some of the more complex features would impose a high performance penalty on restricted systems. That is the reason why its specification does not require that all components implement all of its features. The specification is actually a set of specifications for multiple well defined concepts which may or may not be implemented by the components. The specifications can be organized as increasing *levels of control*, i.e. an increasing order of reflective capabilities. The basic control level provides no reflective capabilities, so that even ordinary objects are Fractal components at this level. Next is the *introspection level* providing means of querying the interface of the component. The final *configuration level* allows the inspection of the component structure and its reconfiguration in runtime.

3.1.2 SOFA

The SOFA (SOFtware Appliances) [4] component model represents component applications as a hierarchy of components. A component can be either *composed* from multiple subcomponents, or *primitive* with no subcomponents. All functionality of the component application lies in the primitive components.

The external interface of a component is defined in its *frame*. The frame specifies its provided and required interfaces and configuration properties. The internal structure of a component is defined as its *architecture*. The architecture specifies the subcomponents of the component and how are they interconnected. Connections between the subcomponents are defined via *interface ties*. There are several types of ties - *binding* interconnects two subcomponents, *delegation* interconnects a *provides interface* of the composite component with a provides interface of a subcomponent, *subsumption* connects a *requires interface* of a subcomponent with a requires interface of the composite. SOFA applications are described via Component Definition Language (CDL) which defines their frames and architecture.

The communication between components is realized by *connectors*. The connectors provide all of the communication and middleware specific functionality, so that the ordinary components implement only the business logic of the application. Connectors separate the business logic from the component hierarchy specific details.

3.1.3 SOFA 2

SOFA 2 [5] is a successor to the SOFA component model. It uses the core component model of SOFA and enhances it in multiple ways. The component model is described via its meta-model and supports dynamic reconfiguration of the component hierarchy in runtime. Connectors are its core part and it uses aspects to separate the non-functional parts of the components from its business logic.

3.2 Model for Heterogeneous Component Applications

The data models of the OMG D&C specification are *platform independent*, which in the terms of the specification means that they are component model independent. The specification expects to be enhanced or refined for specific component models by modifying the platform independent model and thus creating a *platform specific model*, thus using the model driven development paradigm. A platform specific model for the CORBA Component Model is provided in the specification.

We use the term heterogeneous component applications to describe component applications which can use multiple component models for its components. The OMG D&C specification does not address this concept directly. The platform specific models created from the specification's platform independent model can support only a single component model. We are using the specification to support heterogeneous component applications without creating platform specific models. Thus we need a common component data model capable of describing heterogeneous component applications imple-

mented in component models with reasonable properties (such as SOFA and Fractal). We use the capabilities of the platform independent model to deploy heterogeneous component applications. We store the metadata required by the most common component models in the platform independent model of the specification, for example as configuration properties.

As we have seen in section 2.2, the component assemblies in the component data model defined by the OMG D&C specification are purely virtual. They cannot be instantiated because they do not have any associated artifacts. Because they cannot be instantiated, they do not impose requirements on target nodes.

Several component models require assemblies to have an instance at runtime. Fractal which is used as a proof of concept in our work is one of such component models. The absence of instantiable assemblies in the component data model might cause serious problems in our implementation. Because of this we slightly modify the component data model by allowing a component implementation to have both an assembly implementation and a monolithic implementation at the same time, thus creating a *hybrid assembly*. A hybrid assembly can have subcomponents, but it also contains information about its artifacts and requirements. We describe the impact of this change to the component data model on our implementation in section 6.4. The implementation of Fractal support for hybrid assemblies is much simpler and consistent with the planning process. On the other hand the implementation of Fractal support for purely virtual assemblies is highly complex and duplicates much of the planner's work.

3.3 Deployment Runtime

The execution data and management model of the OMG D&C specification is implemented in a deployment runtime described in [7]. The deployment runtime supports the deployment of heterogeneous component applications. Its support for specific component models is extensible via plugins.

The deployment runtime uses the platform independent execution data model without significant modifications. Component model specific metadata are stored via facilities provided by the execution data model, e.g. configuration properties and connections. These facilities are powerful enough to store the metadata required by most component models. The following sections describe the deployment plan metadata required by the Fractal and SOFA component models. Our planning tools need to fill-in these metadata automatically.

Fractal

So far only the Fractal component model is supported by the deployment runtime. The Fractal component model support in the deployment runtime imposes some requirements on the deployment plan which are elaborated in this section.

Component assemblies in Fractal are instantiated. This is a significant difference from the OMG D&C specification, in which the component assemblies are purely virtual with no instances in runtime and serve only to create a better organized logical structure of the component application. The fractal assembly instances provide controller interfaces which can be used for the management of the components. The controllers can configure the assembly and its subcomponents. They can modify the connections between the subcomponents and change the component hierarchy. Moreover, they control the startup and lifecycle of the components. Fractal assemblies can also have their own business logic, so not all functionality is implemented only by the subcomponents. A Fractal subcomponent can be shared by multiple assemblies, thus it can have multiple parent components.

The deployment plan doesn't express the component hierarchy directly, so such information must be stored explicitly by other facilities provided by the deployment plan. The deployment runtime supports two ways of specifying the hierarchy of Fractal components, by special configuration parameters of component instances or by special connections between the parent and child components.

To describe the component hierarchy using the configuration properties, an `InstanceDeploymentDescription` must contain the properties named `parentComponent` and `childComponent`. The values of these properties are the names of the parent or child component instances. Multiple configuration properties with these names can be present in the same instance to store the names of multiple child or parent components. By specifying multiple parent components, a component shared between multiple assemblies is defined. The other way of describing the Fractal component hierarchy is by defining special connections which interconnect special ports of the components. The ports are marked with a configuration property named `type` with the value `parentHood` or `childHood` to distinguish the type of the component relation.

The controllers of a Fractal component differ for a simple monolithic component and for an assembly. Again, because the deployment plan does not support hierarchical components, this information is stored as an execution parameter of a `MonolithicDeploymentDescription`. The execution parameter is named `controllers`, and can have one of the values `primitive` (monolithic component), `composite` (purely virtual assembly) or `hybrid` (assembly with its own business logic). If the parameter is omitted, `primitive` component type is assumed by default.

Each component instance must have a description of its interface provided in a `ComponentInterfaceDescription`. Additional requirements are placed on the interface description. The `specificType` must contain the name of the class implementing the component (can be omitted for composite components because they do not have their own business logic). Each port defined in the interface must have a defined `name`, `specificType` with the name of the interface of the port, and the direction of the port in `provider`. Connections in Fractal can connect only two ports, i.e. they only have two ends (the OMG D&C specification supports more general connections between an arbitrary number of ports with any direction). Thus the `exclusiveUser` and `exclusiveProvider` flags of the ports must be set to `true`.

Fractal imposes additional requirements on the interfaces of the components. To support delegation of the assembly ports onto its subcomponents, special internal ports are created for each external port. Delegation is done by connecting the inner port to a port of a subcomponent. The inner ports have the same names as the external ports. They are identified by an execution parameter of the port named `type` with the value `internal`.

SOFA

The SOFA component model is not implemented in the deployment runtime [7], thus the planning tools will not support it too. However, the requirements for SOFA were elaborated in the work [7]. SOFA places less requirements on the deployment plan, as it does not require explicit representation of the parent-child relationships between components nor does it need to distinguish between monolithic and composite components. On the other hand it requires special connections representing propagation of configuration properties to subcomponents and an identification of port type (connector, delegation or subsumption, delegation or subsumption internal).

Altogether the requirements of SOFA are simpler than Fractal's, thus adding support for this component model in planning tools would be easier.

3.4 Connectors

Component instances use connections between ports to communicate with other instances. In many cases the communication is done over a computer network, thus communication middleware must be used. The middleware handles the network communication between the components. The middleware must support the component model and technologies used by the component instances.

Our situation is complicated by the fact that we support heterogeneous components applications. The components on each side of the connection can use a different component model, technology, programming language

etc. For the purpose of this work we assume that this problem can be solved by generating connectors [15].

The work [15] proposes that communication between components is handled by connectors. Each connector consists of multiple connector units, which handle the component model specific part of communication with their respective components and the middleware specific part for communication with the other connector units. This approach removes the middleware specific code from the components and moves it to the connectors, thus the components handle only the business logic. Components communicate with their respective local connector unit the same way as with a local (not separated by a network) component. The connector unit routes the communication to remote connector units.

Connectors do not have to be written manually and packaged in the component repository. A connector framework can generate connector code on demand if integrated with the deployment framework including our work.

Our work is designed to be extensible in such a way that support for connectors can be added to it in the future. This support would use the same mechanics which are used to add component model specific metadata to the deployment plan.

4 Goals Revisited

In the previous sections we have analyzed the OMG D&C specification and the requirements of the deployment runtime. Therefore we can re-visit the goals to be achieved by this work. We can view the goals from the standpoint of two views, the end user and the developer. The user utilizes the planning tools to create a valid deployment plan and interacts with the other parts of the deployment framework. The developer further enhances the planning tools to support additional features of the OMG D&C specification and the future work on our deployment framework. The goals presented in both views do not strictly belong to their respective views, but the views present the main focus of the goals.

4.1 User View

Graphical interface

Planning is a highly interactive task. Users must be able to make planning decisions in a user-friendly way. This requires a graphical user interface (GUI). The users want to influence the planning decisions, so they must be able to view the information necessary for that. Specific views of component metadata and of the domain must be available in a well structured form.

Computer Assisted Planning

Creating a valid deployment plan is a complex task. It involves finding hosts that match the requirements of the components. The requirements are non-trivial and their semantics can be defined by the deployment framework. Some of the requirements allocate resources thus decreasing their availability. On the other hand, the user is usually not interested in the specific planning of most components of his component application. He focuses his attention on the most critical components, and does not pay much attention to the other components as long as they are planned on a valid node. Thus an automated or a semi-automated tool is needed to assist the user in creating a valid deployment plan in a user-friendly way. The task of creating a planning is complex and can take significant time, so it should be interruptible with the ability to continue with it.

Extensibility

The deployment framework is a complex software system. It provides numerous interfaces and facilities which should be available to the user. This work focuses on planning, and creating a GUI with full support of the deployment workflow is beyond its scope. On the other hand it would be beneficial if the

GUI could be easily extended in the future to fully support other parts of the deployment framework and its workflow.

4.2 Developer View

Extensible Planning

The planning tools will use an algorithm to assist the user in creating a valid deployment plan. An algorithm is required so that deployment plan can be found automatically or with minimum user input. Component applications are used in different contexts and in a wide range of problems. Thus, planning component applications is a big domain, and a "one-fits-all" algorithm may not be practical. Demands placed on the algorithm can change from one deployment to another. Thus an extensible algorithm is needed. It should be easy to modify the behavior of the algorithm.

Heterogeneous Component Applications

Heterogeneous component applications impose requirements on the deployment plan, as we have seen in the analysis of the deployment runtime in section 3.3. The deployment plan must contain information which is either missing in the metadata of the planned component, or it can be generated automatically. Components contained in the deployment plan may require component model specific metadata or tranformations of the deployment plan. The planning tools should fill or modify the deployment plan with the required information automatically. This must be extensible to support additional component models.

The component data model defined by the OMG D&C specification does not address heterogeneous component applications. We must find a way to store the metadata required by various component models in the component data model, possibly with as few modifications of the model as possible.

Integration

Our deployment framework is a continuous work-in-progress. Additional features may be added to it in the future. Some features like connectors implement functionality not defined in the OMG D&C specification. The planning tools should be heavily integrated with the deployment framework and they must be easily extensible to support additional features of the framework.

5 Planner

The *planner* is the non-GUI part of this work. It's the core algorithm which tries to find a valid deployment plan and its supporting tools. This section outlines the design of the planner. There are several key points of the algorithm that guide its design:

- **User interaction:** The user must be able to influence and review the results of the algorithm. User interactions are implemented in a GUI, which is integrated with the planner. The GUI communicates the user's input and preferences to the planner.
- **Automation and planning heuristics:** The planning algorithm searches for a valid deployment plan by itself. Finding a valid deployment plan is a complex task, thus we use several heuristics.
- **Extensible:** The heuristics and decision points of the algorithm can be modified and extended. The resulting deployment plan must contain component model specific metadata as required by our deployment runtime. The planner generates such metadata automatically and its support for additional component models is extensible too.

5.1 Planner Design

Goal of the planner is the automatic generation of a valid deployment plan. Its input is the metadata of the planned component, information about the domain and the user input which can influence the planning. While a planning algorithm constitutes the core of the planner, it is unable to create a valid deployment plan by itself. Additional steps are required for that. The basic structure of the planner is outlined in figure 5.

As we have seen in the analysis of the execution data model in section 2.6, the deployment plan represents a flat component application. The flat component application does not support any notion of component hierarchy. On the other hand even the assemblies of the component data model do not contain any information needed to create their instances as can be seen in section 2.2. We have proposed a change to the component data model which allows *hybrid* instantiable assemblies. Hybrid assemblies contain both an assembly of subcomponents and an implementation description necessary to create their instances at runtime. However, we still keep the possibility of assemblies which are purely virtual and serve only as a tool for organizing and composing component applications. The virtual assemblies are removed by *flattening* the component application metadata. Flattening is a process which recursively removes virtual assemblies, and merges some of their metadata with their subcomponents. The result of such operation is the metadata of a

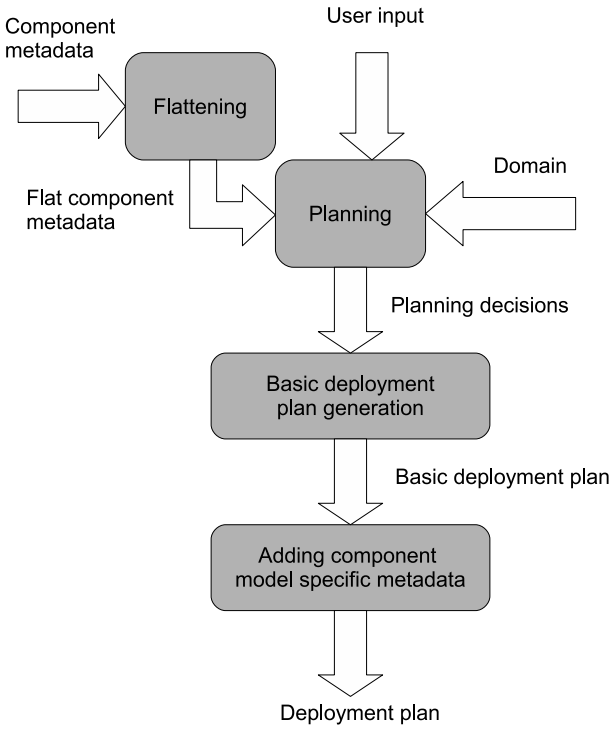


Figure 5: Planner basic design

flat component application which consists only of monolithic components or hybrid assemblies. Functionally the flat component application is equivalent to the original hierarchical application.

The flat component metadata is used by a planning algorithm which tries to find a valid assignment for all components and connections. Assignment of a component is defined by the selection of a node which matches the component's requirements. Similarly connections are assigned onto interconnect. Components and nodes with a found assignment are *assigned*, without an assignment *unassigned*.

A base skeleton of a deployment plan can be created when the planning algorithm successfully finishes. The deployment plan skeleton describes the basic structure of the resulting component application, i.e. its instances, connections, configuration properties etc.

However, the basic deployment plan cannot be executed by the deployment runtime. It still needs to be enhanced with metadata specific for various components models as required by the deployment runtime (as described in section 3.3). Thus the basic deployment plan is modified to suit the requirements of the component models in the final step. The final step is realized by extensible modules which detect components implemented in their respective component model and modify the deployment plan accordingly.

5.2 User Interaction

While the planning algorithm is fully automatic, users of the planning tools still need a way to influence its decisions. The users can have their own requirements on the deployment of the component application or they may have more complex knowledge of the current state of the deployment environment. Users will usually interact with the planner via a GUI which is integrated with the planner. The GUI receives the users' planning decisions and preferences and displays the resulting planning.

There are several usual use cases of user interaction which are implemented by the planner. Most of the time users creating a deployment plan know exactly what they do not want and have a basic idea of what would they prefer. For example, a user wants to specify that a database component must not be assigned on a specific node which is not reliable enough. User's preferences mean that for example he would prefer the application and web server components to be assigned onto another specific node because he thinks that the node will provide them with excellent performance. The preferred node should be taken into account by the planner, but if the node does not satisfy the components' requirements it can be safely ignored and another one can be chosen. Another typical use case is a user specifying a set of allowed nodes for components, for example a high performance node cluster for running resource intensive components.

The planning decisions of the algorithm can be influenced by *node restrictions*. Each component of a component application can have node restrictions associated, in which case these are taken into account by the planner. If no node restrictions are associated with a component then the user gives the planner full control over the component's planning. Node restrictions are represented by three sets of nodes (each of them can be possibly empty):

- **Allowed:** The component must be assigned onto one of the allowed nodes, all other nodes are implicitly forbidden if this set is specified.
- **Forbidden:** The component must not be assigned on any of the forbidden nodes, all other nodes are implicitly allowed if this set is specified.
- **Preferred:** The user would prefer if the component is planned on one of the preferred nodes. But this preference is not mandatory, the planner can ignore the preferred nodes if they do not meet the component's requirements.

Only one of the allowed or forbidden sets should be used, because they implicitly forbid or allow the remaining nodes. Which set is used depends on the use case, e.g. "allow all nodes except these" or "allow only these nodes".

The planner does not support *interconnect restrictions*, which would be analogous to node restrictions. The most typical use cases of user interactions

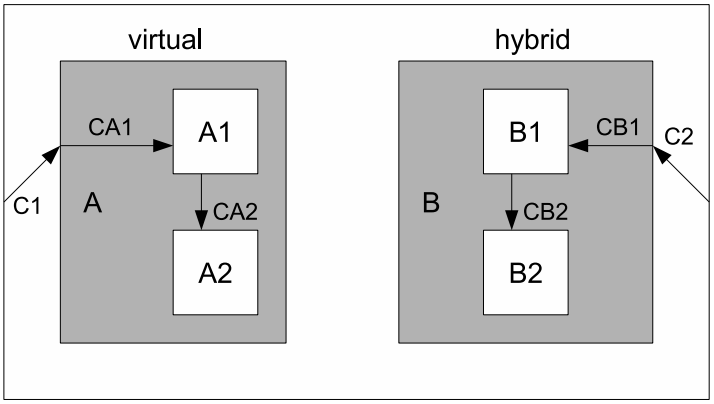


Figure 6: Component metadata before flattening

involve components, and the users do not care much about the connections as long as they are correctly planned. However adding support for interconnect restrictions is simple as the planner is extensible (see section 5.5).

5.3 Flattening

Flattening is a process which pre-processes the component metadata so that they contain only instantiable components. The resulting component metadata represents a flat component with no deep component hierarchy. The flat component is either one monolithic component, or an assembly with subcomponents which are either monolithic components or hybrid assemblies. The hybrid assemblies in the resulting component metadata do not contain subcomponents, these are recursively extracted to the top level assembly.

Overview of the flattening process is provided in figure 6 and figure 7. Figure 6 represents the original, un-flattened component application. It is an assembly which contains two subcomponents, A and B. Both of them are assemblies with two monolithic subcomponents. Assembly A is a purely virtual assembly, while the assembly B is hybrid. The component application contains several connections. Connections C1 and C2 in the top level assembly delegate ports of the top level assembly to ports of its subcomponents A and B. Both assemblies further delegate the connections via connections CA1 and CB1 to a port of one of their monolithic subcomponents. The monolithic subcomponents also have a connection between them (CA2 and CB2).

After flattening, the component metadata has a different structure as can be seen in figure 7. The virtual assembly A is completely removed. Its two monolithic subcomponents became subcomponents of the top level assembly. The hybrid assembly B is left intact, but its monolithic subcomponents were extracted from it and became subcomponents of the top level assembly. The resulting component application is flat, i.e. none of its components have any subcomponents. The flattening process is recursive. It implements a

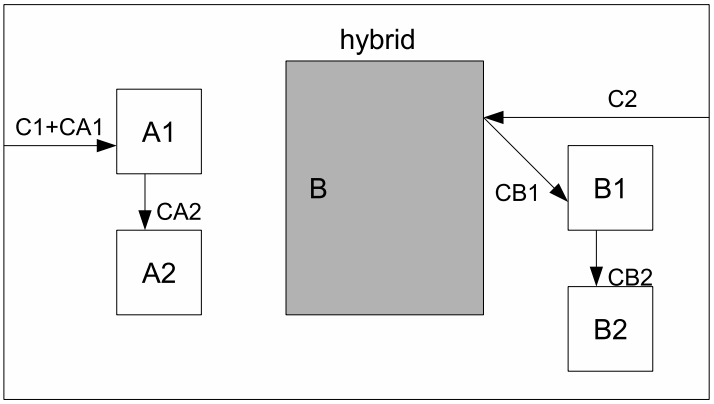


Figure 7: Flattened component metadata

depth-first traversal of the component hierarchy, recursively flattening each component and then backtracking to its parent component.

Connections between components may need to be modified during flattening. Simple connections which only interconnect several subcomponents of an assembly (virtual or hybrid) are extracted from the assembly together with the subcomponents (see connections CA2 and CB2). They are extracted without modification, thus only their locations change. Modification is required when a connection is connected to a port of an assembly, which is further delegated to a subcomponent. If the port belongs to a virtual assembly, then the assembly will be removed during flattening and the connections must be merged. This is illustrated by connections C1 and CA1 in figure 6 which are merged into the connection C1+CA2 in figure 7. Merging of connections joins their requirements and ports to which they are connected. Merging of such connections is not needed for hybrid assemblies, as their instances are created in runtime and the connections can interconnect the instances. Such connections are only extracted from the hybrid assembly and connected to its ports "from the outside" without further modifications.

Assemblies can contain configuration property mappings, which propagate their configuration properties onto their subcomponents. These mappings are also merged similarly to connections.

The resulting flattened component application is used by the planning algorithm to find a valid deployment plan, and it is used to generate the basic deployment plan (without component model specific metadata). The planning algorithm thus only uses the direct subcomponents of the top-level assembly and ignores any possible subcomponents deeper in the component metadata. Because of this, there is not need for subcomponents of hybrid assemblies to be removed during their extraction, they are copied to the parent component of the hybrid assembly. Thus the information about component hierarchy remains in the component metadata during planning, and

the planning algorithm may use it (e.g. for the planning of the hybrid assemblies relative to their subcomponents).

The flattening process stores information about the modifications done to the component metadata. Other parts of the planner can use this information later (see later section 6.4.2 for an example).

5.4 Planning and Scheduling

Planning and scheduling is a branch of artificial intelligence [8] [9] [10] which studies the sequences of actions required to achieve a desired goal, and the allocation of resources for such actions.

Problems solved by planning algorithms transform a domain from its starting state into a desired target state. The domain has a pre-defined set of actions which can modify its state, so the goal of the planning algorithms is to find which actions and in what sequence transform the domain to the desired target state. The planning algorithms ignore time and resource requirements of the actions.

Scheduling solves the allocation of planned actions in time and space. It allocates resources for the actions, which were chosen and possibly ordered during planning. Some plans of actions cannot be successfully scheduled. Sometimes planning and scheduling are not solved separately, especially when a big proportion of plans cannot be scheduled correctly.

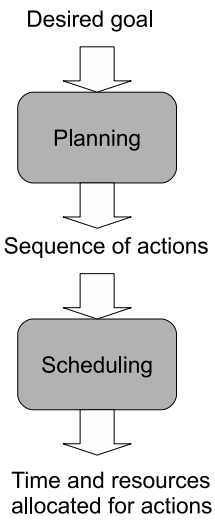


Figure 8: Planning and scheduling

Successful and complex algorithms were invented to solve the planning and scheduling problems. They often use formal languages or systems as *Constraint programming* [11] to describe the problem and its domain.

Unfortunately, the planning problem solved by our work is more general than the problems solved by the usual planning and scheduling algorithms. Our planning algorithm must satisfy the requirements of components with the resources of the domain. The resources are not compatible, as the OMG D&C specification gives us the possibilities to define custom resources with various semantics. So for example, some resources are just compared with the requirements ("CPU speed must be at least 1000Mhz"), some need to be allocated ("I need 512MB of RAM") and some are just tested for existence ("Java Runtime Environment must be installed"). Even resources with the same semantics (e.g. "resource value must be greater than or equal to the requirement") are not compatible because of varying measurement units (CPU speed vs memory throughput). Components can also impose requirements on the resulting

planning via locality constraints which are yet another kind of requirements.

Because of the generality of the component and target model, advanced planning and scheduling algorithms are not suitable for our planning problem. We use a custom simpler backtracking algorithm. Our planner needs to solve relatively small planning problems (e.g. tens or a few hundreds of components, tens of target nodes) and the user can help it if needed, so an optimal planning algorithm is not really necessary. A custom simpler algorithm allows us to modify and extend it easier.

5.5 Algorithm

Algorithm chosen to solve our planning problem is depth-first-search, i.e. backtracking, which is one of the simpler algorithms used for planning and scheduling. The algorithm searches the state space defined by the *assignments* of components onto nodes and connections onto interconnects. The desired target state is such a planning in which all components and connections are assigned on a node or interconnect. The target planning must be valid, so that the requirements of the components and connections are met by the resources present in the domain, without over-allocating the resources. Of course the target planning must respect the logical structure of the component application by assigning on such nodes and interconnects that the components can be successfully interconnected.

The algorithm searches the state space of all possible assignments in recursive steps. The first step begins with an initial state represented by an empty planning (no nodes or interconnects selected for components and connections). In each step, the current state is evaluated whether it is the final state in which all components and connections have a valid planning. If not, the current state is modified by planning a single *unassigned* component or connection on a valid node or interconnect, and the algorithm recurses into a deeper step. If the following step succeeds, then the current step succeeds too and control is given back to the previous step. If the following step does not find a valid planning, the modification of the current step is rolled back and another planning is tried for a component or connection (i.e. another state is examined). If the current step cannot find any valid planning, then it returns back with failure.

A basic overview of the algorithm can be presented as pseudocode in code listing 1. The pseudocode demonstrates one step of the algorithm. At the start of the step it checks whether there is any work left to do. If all components and connections are assigned (i.e. a valid node or interconnect was selected for them in previous steps), then the step returns successfully.

A matching phase follows if there are any unassigned components or connections left. For each unassigned component (or a connection, respectively), all nodes (connections) which match its requirements are found. By require-

Algorithm 1 Planning algorithm

```
if everything is assigned then
  return true
end if
match components
match connections
if any component or connection cannot be planned then
  return false
end if
if all components assigned then
  if plan connections then
    return true
  else
    return false
  end if
end if
for unassigned component do
  for matching node do
    allocate resources for component requirements
    if deeper step successfull then
      return true
    else
      rollback resource allocation
    end if
  end for
end for
return false
```

ments here we mean *planning requirements* in general (i.e. locality constraints), not just the requirements satisfied by resources provided by nodes or connections. This matching process is the first *extension point* of the algorithm so that it can be improved to support additional planning requirements given by additional technologies.

If there is no node which matches the planning requirements of some component (respectively an interconnect for a connection), the planning step fails. This ensures the fail-fast behavior of the algorithm, which detects dead-ends in the state space search as soon as possible.

The algorithm is focused on planning all of the components first, then the connections. This behavior simplifies the algorithm and it is natural considering the typical use case (the domain consists of many computer nodes connected to one or just a few interconnects). Planning of connections is simplified compared to the planning of components and its heuristic are not

currently extensible, but the idea is similar. Each unassigned connection is tried to be assigned on each one of its matching interconnects in succession, and then a deeper step is executed for another connection. If the deeper step fails, the connection is assigned on another one of its matching interconnects etc.

The main part of the algorithm step tries to select a valid node for an unassigned component. One assignment modifies the current state being examined by the algorithm, and sends the algorithm into one direction of the search for the final valid state. The algorithm iterates over the currently unassigned components. For one currently selected unassigned component a node is selected from the set of nodes which match the component's planning requirements. The selected unassigned component is assigned onto the selected node, and the algorithm recurses deeper into another step. If the deeper step is successful and thus finds a valid planning of all remaining unassigned components and connections, the current step returns successfully. If the deeper step fails then the selected assignment is rolled back (with the allocated resources freed) and another assignment is tried for the currently selected unassigned component. If an unassigned component cannot be successfully planned on any of its matching nodes, another unassigned component is selected and the operation is repeated. The step fails if none of the unassigned components can be successfully planned.

The order of iteration over the unassigned components strongly influences the direction in which the planner searches the state space. It is a natural decision point where a heuristic should be used. The *component selection* extension point of the planning algorithm allows modification of the heuristics.

For each unassigned component, nodes are selected from the set of its matching nodes. The order of this *node selection* also influences the behavior of the algorithm, thus it is an additional extension point.

The time cost of our planning algorithm depends on the deployment scenario. We define *Comp* as the number of components in the component application, *Conn* as the number of connections, N as the number of computer nodes in the target domain, I as the number of interconnects and the number of explored assignments as the time cost of the algorithm. The general, potentially very complex, deployment scenarios can involve resource allocation (e.g. memory size) and can cause backtracking. Thus the time cost of the algorithm in general deployment scenarios is exponential:

$O(N^{Comp} I^{Conn})$

Simple deployment scenarios involving only comparison of requirements and resources without allocation (e.g. CPU speed) have a linear time cost because no backtracking is needed:

$O(Comp + Conn)$

Real life usage scenarios can present planning problems from the simplest

to the most complex. Further analysis and benchmarking of our algorithm is required to determine the impact of our planning heuristics. However, such complex benchmarking is out of the scope of this work and presents a topic for future work.

5.5.1 Component and Connection Matching

Near the beginning of each step of the planning algorithm, all nodes which match the requirements of unassigned components, are selected from the set of all available nodes during the *matching phase* (similarly interconnects for unassigned connections). The requirements are general planning requirements imposed by the planning process, not just the components' and connections' requirements on the resources available at the target nodes and interconnects. Locality constraints, component application structure etc. are all examples of planning requirements.

The matching process is one of the *extension points* of the planning algorithm. Matching is done by *matchers*. There are two kinds of matchers:

- **Component matcher:** Matches the planning requirements of components. It decides whether a specific node is valid for a specific component.
- **Connection matcher:** Matches the planning requirements of connections. It decides whether a specific interconnect is valid for a specific connection.

In the beginning of each step of the planning algorithm, all nodes which match the requirements of unassigned components are found (similarly interconnects for unassigned connections). The matching phase iterates over all unassigned components and for all nodes of the domain it determines whether the node matches. The decision is made by querying all component matchers whether the node matches the component's planning requirements. The node matches only if all matchers return success, and the matching of the node stops when the first matcher returns failure. Similar process is used for connection matching.

All information required to make the matching decisions are available to the matchers:

- **Component or connection metadata:** Metadata of the unassigned component or connection are passed to the matcher.
- **Node or interconnect description:** Complete description of the node or interconnect being matched is available to the matcher. The description contains all information stored in the target manager (see section 2.4 for a description of the available information).

- **Planning state:** The current state being searched by the planning algorithm is passed to the matcher. In other words, the current partial planning result is passed to the matcher, so it can decide based on the current assignments of the already assigned components and connections.

The matchers provide the ability to support future technologies in the planner, but are also used to provide the core functionality required by planning as defined by the OMG specification. The core functionality is provided by default matchers which are supplied with the planner.

5.5.2 Component Selection

The planning algorithm searches all possible plannings (i.e. states) for the one which is valid. The current planning state is modified in each step of the algorithm by assigning one component onto one valid node. The component is chosen from a set of components which are not assigned yet. If a component cannot be successfully assigned on any node, another component is chosen from the set of unassigned components. The order in which the components are selected is called *component selection*. Component selection is one of the decision points, or heuristics, of the planning algorithm which influence the way in which the current planning state is modified. It is also an extension point allowing future improvement of the planning algorithm.

Component selectors implement this extension point. One component selector must be active for the planning, and its goal is to create a total ordering of currently unassigned components. Component selectors have the following information available to them:

- **Matching result:** Result of the matching of components is the primary source of information used by the component selector. The result contains all the nodes which match the planning requirements of each unassigned component.
- **Planning state:** The current planning state, i.e. the current partial planning result.

5.5.3 Node Selection

The currently searched planning state is modified by planning an unassigned component onto one of the nodes which match its planning requirements. If one such planning fails, another node is tried for the unassigned component. The order in which the nodes are selected from the set of matching nodes is called *node selection* and it is one of the decision points, or heuristics, of the algorithm which influence the way the current planning state is modified.

It is also an extension point allowing future enhancement of the planning algorithm.

Node selectors implement this extension point. One node selector must be active for the planning, and its goal is to create a total ordering of nodes which match the planning requirements of the currently selected unassigned component. Node selectors have the following information available to them:

- **Matching result:** The matching result contains all nodes matching the requirements of the one component which is currently being planned.
- **Planning state:** The current planning state, i.e. the current partial planning result.

5.6 Heterogeneous Component Applications

The basic design the planner consists of multiple steps, with a valid deployment plan being the final result if the planning is successful. When the core planning algorithm successfully finishes, a basic skeleton of a deployment plan can be generated. The skeleton describes the overall structure of the deployed component application but is does not contain component model specific metadata required by the deployment runtime (as described in section 3.3). The component model specific metadata are required by the deployment runtime because it uses a common execution data model (as defined by the OMG D&C specification) for the deployment of components implemented in various component models. The common model does not express all concepts used by some component models, such as component hierarchy. We store these concepts in the common execution model by other means, represented by component model specific metadata such as special connections or configuration properties.

Thus in the final step of the planning, the component model specific metadata are automatically filled-in. This is accomplished via *deployment plan metadata providers*. A metadata provider can modify the basic deployment plan in any way required. The modification is usually defined by the requirements of the deployment runtime's support for a specific component model. The most common modification of the deployment plan is the filling in of special configuration properties (e.g. the properties which describe the parent-child relationship of Fractal components, see section 3.3). However, even more complicated modifications may be needed, as changes to component interfaces (as is the case with Fractal), generation of special interconnects etc.

The basic idea of a component model metadata provider is that when it detects some specific information in the component metadata (e.g. an identifier of its supported component model) it modifies the deployment plan

in some custom way. The information required in the component metadata is arbitrarily defined by the metadata provider, the planner does not pre-define them. Thus the metadata providers impose requirements on the component metadata, not the planner.

The planner runs all active metadata providers in succession, with each of them modifying the deployment plan arbitrarily. Each of the deployment plan metadata providers then uses its own way of detecting which components are implemented with its component model, and modifies the deployment plan accordingly. This enables us to plan heterogeneous component applications, with the metadata providers imposing requirements on the component metadata. Of course the metadata providers must modify the deployment plan in such a way which is compatible with other metadata providers' modifications. This is not a big issue, as the metadata providers usually only need to modify the components implemented in their respective component model and ignore other components, thus their modifications are localized.

Deployment plan metadata providers are the means of supporting heterogeneous component applications in the planner, thus they are an additional extension point. However they are much more general than just that, as they can modify the deployment plan in any way. This can be used to add support for additional technologies to the planner, such as connectors.

6 Planner Implementation

The planner is implemented in Java as a library (JAR file) which can be embedded or integrated within additional tools, especially graphical front-ends. The implementation contains the core planning algorithm and the framework supporting it (flattenning, base deployment plan generation etc). Basic means of extensibility of the planner via plugins are provided. The plugin architecture is also used to implement the core functionality and heuristics of the planner. An overview of the classes used in the planner implementation is provided in figure 9. The classes are further elaborated in the following sections.

Java 1.5 was chosen for the implementation for interoperability with the other existing parts of our deployment framework and for the wealth of available tools.

One of the goals of the planner is to be integrated within additional tools, like the GUI provided with this work. The tools interact with the user and use the planner to create a valid deployment plan automatically. However, the planner is also distributed with a set of simple tools that can be used to test and develop the planner further without a graphical user interface.

6.1 Algorithm Implementation

Interface

An interface for user interaction with the planner is defined in the `Planner` interface. The interface is not designed for multiple alternative implementations of the planning algorithm, but mainly as means for separation of concerns as it focuses on the user interaction. The interface specifies one method `plan` which starts the planning algorithm and returns a description

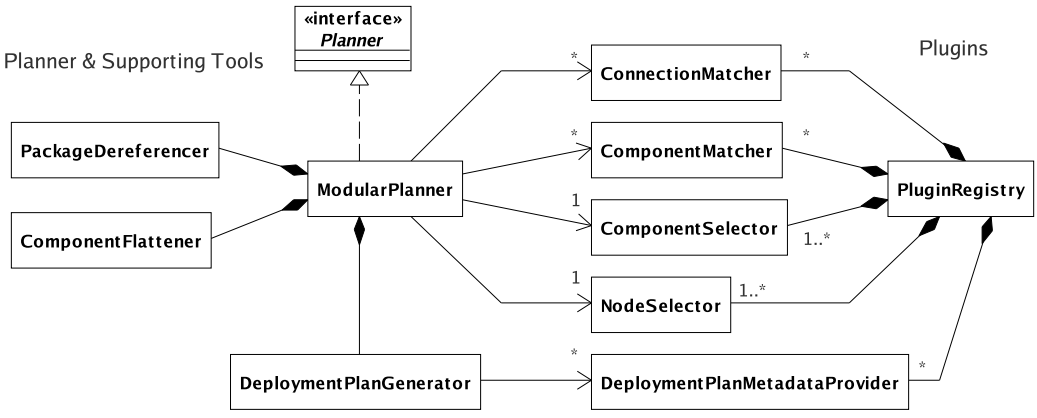


Figure 9: Planner implementation classes

of its result as a `PlannerResult`. The result contains the following information:

- **Result status:** Describes whether the planner successfully found a deployment plan or not.
- **Deployment plan:** Contains the generated deployment plan in case the planner finished successfully.
- **Resource allocation:** Description of resources allocated by the deployment plan. It is basically a copy of the resource allocation stored in the deployment plan for easier access.

The only parameter of the `plan` method is `PlanningMetadata`. The planning metadata contain information mostly used for user interaction:

- **Node restrictions:** Specifies restrictions on nodes assigned to components, i.e. allowed, forbidden and preferred nodes.
- **Planning control:** Allows interruption of the planner algorithm.
- **Planning progress:** Describes the progress of the planning in terms of the number of currently assigned and unassigned components or connections. These numbers can both rise and drop during the course of the planning, as the recursive planning algorithm may need to backtrack, i.e. cancel multiple plannings.

The core of the planning algorithm (which is described in section 5.5) is implemented by the `ModularPlanner` class which implements the `Planner` interface. The name of the class implies that the planning algorithm is designed for extensibility. The modular planner is intended to be used for one run of the planning algorithm, which simplifies its design. Thus the metadata of the planned component application and the metadata of the target domain are passed to the modular planner as constructor parameters.

Workflow

The modular planner's implementation of the `plan` method is composed of multiple steps:

1. **Package dereferencing:** Metadata of the planner component can contain references to additional packages via configuration specialization or via a reference to another package in the metadata repository (see section 2.2). These references are imported into the component metadata. References to other packages in the metadata repository are currently unsupported for simplicity. This functionality is implemented in the `PackageDereferencer` class.

2. **Flattening:** Metadata of a hierarchical component application are flattened into a flat component application (via a process described in section 5.3). The flattening is implemented by the `ComponentFlattener` which recursively traverses the component metadata and flattens them on the way back. Original hierarchical metadata are backed-up for further use during the generation of the deployment plan.

3. **Initialization:** Various internal data structures are initialized for use during the planning. The data structures contain the current planning state and pre-processed information about the component application created for quick access during the algorithm (e.g. a list of connections which are connected externally to a component).

4. **Planning:** The recursive planning algorithm is launched. If successful, its result contains assignments of components onto nodes and connections onto interconnects. The algorithm is implemented by the recursive `doStep` method which takes the current planning state as parameter.

5. **Deployment plan generation:** If the planning finished successfully, a deployment plan is generated in two steps. The first step generates a basic skeleton of a deployment plan. The second step modifies the deployment plan with component model specific metadata.

The `doStep` method implements the planning algorithm. Its structure is very similar to the pseudocode describing the planning algorithm in section 5.5. It executes the component and connection matching to validate planning requirements. Component and node selectors are used to guide the flow of the algorithm. The current state of the planning is described in the `PlanningContext`, which stores the following information:

- **Assigned components and connections:** Planning decisions for components and connections which are already assigned onto a node or interconnect. The planning decisions are stored in both directions, i.e. what node was selected for a component and also what components are assigned to a node. Both directions are potentially usefull, especially for some planning heuristics.

- **Unassigned components and connections:** List of components and connections which are not assigned to a node or interconnect.

- **Matching preferred nodes:** Preferred nodes of components (as specified in their node restrictions) which also match their planning requirements.

- **Planning metadata:** The planning metadata are used for user interaction with the planner. The planner checks the metadata for an indication that it should stop at the start of each step (i.e. it is interrupted externally by the user).

Logging

Debugging and testing the planner is a complex task. To ease this task the planner provides thorough logging information. Logging is implemented with the widely used log4j [25] framework, which is also used in the other parts of our deployment framework. The planner uses several independent loggers to provide debugging information for different aspects of its workflow, especially component and connection matching and the core planning algorithm. Each of these loggers can be independently configured depending on the developer's or user's focus. Logging via log4j is implemented carefully, as it can severely impact performance of the algorithm. Complex logging messages are created with heavy use of `String` concatenation. If these messages are passed as parameters to the log4j logging methods, the `String` concatenation is performed before the test whether the message is logged or not. Because of this we test the current log level manually before creating the log messages. This is a general problem with logging frameworks similar to log4j, as using them in performance constrained environment requires the use of additional code which makes the resulting code less comprehensible.

6.2 Aglorithm Extensibility

The planning algorithm is designed to be extensible in several points. This extensibility allows us to improve the algorithm with better heuristics and with support for future technologies. Also, the extensibility mechanisms are used to implement the core functionality of the algorithm as required by the OMG D&C specification.

All the extension points of the algorithm are implemented by classes extending pre-defined abstract classes. The abstract superclasses form a simple framework and serve as an interface to the implementations. Implementations of the extension points are registered in a simple plugin registry (described in later section 6.5).

6.2.1 Component Matching

Component matching is the process of finding nodes which match the planning requirements of unassigned components, as described in section 5.5.1. It is performed by component matchers, which are one of the means of extensibility of the planning algorithm. Each component matcher tests whether a node matches some specific kind of planning requirement of a component.

Component matchers must extend the `ComponentMatcher` class. The class defines the interface of the matcher and provides several utility methods:

- **isMatching:** This abstract method must be implemented by the subclass. It evaluates the planning requirement which is tested by this component matcher. Its parameters are the metadata of the unassigned component, metadata of the node and the current planning state (i.e. `PlanningContext`). The planning context can be used to perform matching which is based on the current partial planning result. Result of the `isMatching` method is stored in `ComponentMatchingResult`, which describes whether the matching was successful and potentially contains a description of allocated resources.

- **createTrue:** Creates a `ComponentMatchingResult` which describes a successful matching result. A variant of this method stores a description of allocated resources in the result.

- **createFalse:** Creates a `ComponentMatchingResult` which describes an unsuccessful matching result.

Component matchers are also used to implement the core functionality of the planner via default component matchers. These matchers are supplied with the planner and are enabled by default. Default component matchers are:

- **Node restrictions:** The planning can be influenced by the user via node restrictions (see section 5.2). The node restrictions can specify which nodes are allowed, forbidden and preferred for each component. The *node restrictions component matcher* enforces these restriction for the components.

  For each component and node the restrictions are evaluated as follows:

  1. if the component has no node restrictions specified, the node matches.
  2. if the node belongs to the preferred set of nodes, it matches.
  3. if the node belongs to the allowed set of nodes, it matches.
  4. if the node belongs to the forbidden set of nodes, it does not match.

- **Resources:** The *resources component matcher* determines whether all requirements of the component are satisfied by the target node. The requirements of a component can be defined by its monolithic implementation or by its artifacts. The artifacts can recursively depend on additional artifacts which may specify their own requirements. Thus the

requirements of a component are defined as the sum of all requirements of its monolithic implementation, artifacts and artifact dependencies. A hybrid assembly can also impose its own requirements because it contains not only an assembly implementation, but also a monolithic implementation.

For each requirement a resource which satisfies it must be found on the target node. The satisfying resource must be of the same type as specified in the *resource type* of the requirement (e.g. "memory"). The OMG D&C specification provides us with the ability to define the semantics of resources by specifying a *satisfier kind*, which describes the way in which the resource satisfies requirements. Some of these satisfier kinds specify that the value of the resource is modified when it satisfies a resources, i.e. the resource is allocated for the requirements (i.e. "memory size"-like resources). Such allocations must be stored by the planning algorithm, thus they are returned by the resource component matcher. If no resource which satisfies a requirement is found, then the node does not match the component. On the other hand if a component does not specify any requirements, then all nodes match its requirements. Multiple resources can satisfy the same requirement, and this matcher selects the first one.

- **Locality constraints:** Enforces the locality constraints specified in the component metadata (see section 2.2).The component being matched can take part in multiple locality constraints, with each of them defining a relation between the constrained components. The locality constraints are of multiple types, which can be separated into two groups by their relevance for the planning algorithm: *same node* and *different node* constraints. The *same node* group has several subtypes which are irrelevant for the planning algorithm because they specify the process separation of the components on the same target node (see section 2.2).

  For each component and node the locality constraints are evaluated by the *locality constraints component matcher* as follows:

  - if the matched component does not take part in any locality constraint, then the node matches
  - if the matched component takes part in a *same node* locality constraint, then all other components constrained by the same constraint are checked. If a constrained component is already planned on a node, then only the same node matches. If no other constrained component is planned yet, then any node matches.
  - if the matched component takes part in a *different node* locality constraint, then all other components constrained by the same

constraint are checked. If a constrained component is already planned on a node, then only a different node matches. If no other constrained component is planned yet, then any node matches.

- **Planned connections accessibility:** If any connection which is connected to the matched component is already planned, then only nodes which are connected to the planned interconnect of the connection are matching the component. Thus the *planned connections component matcher* enforces the correct logical structure of the component application, such that all components can be successfully interconnected. Currently this component matcher is redundant, because all connections are planned only after all components are already planned. But the matcher is implemented for possible future modifications of the planning algorithm.

6.2.2 Connection Matching

Connection matching is analogous to component matching, i.e. it is the process of finding interconnects which match the planning requirements of unassigned connections, as described in section 5.5.1. It is performed by connection matchers, which are one of the means of extensibility of the planning algorithm. Each connection matcher tests whether an interconnect matches some specific kind of planning requirement of a connection.

Connection matchers must extend the `ConnectionMatcher` class. The class defines the interface of the matcher and provides several utility methods:

- **isMatching:** This abstract method must be implemented by the subclass. It evaluates the planning requirement which is tested by this connection matcher. Its parameters are the metadata of the unassigned connection, metadata of the interconnect and the current planning state (i.e.`PlanningContext`). The planning context can be used to perform matching which is based on the current partial planning result. Result of the `isMatching` method is stored in `ConnectionMatchingResult`, which describes whether the matching was successful and potentially contains a description of allocated resources.

- **createTrue:** Creates a `ConnectionMatchingResult` which describes a successful matching result. A variant of this method stores a description of allocated resources in the result.

- **createFalse:** Creates a `ConnectionMatchingResult` which describes an unsuccessful matching result.

Connection matchers are also used to implement the core functionality of the planner via default connection matchers. These matchers are supplied

with the planner and are enabled by default. Default connection matchers are:

- **Resources:** Connections between components can define requirements on the resources available on interconnects. The *connection resource matcher* matches only the interconnects which match the connection's requirements, similarly to the component resource matcher. Resources satisfying the connection's requirements are searched on the interconnect and their allocation is stored if needed.

- **Planned nodes:** A connection can be connected to multiple components. Some of those components may be already planned onto specific nodes. The *planned component connection matcher* matches only those interconnects which are connected to those nodes. As the planning algorithm progresses, this matcher narrows the possible interconnects for connections because their components are being planned onto specific nodes. This connection matcher enforces the correct logical structure of the planned component application.

6.2.3 Component Selection

Component selection is the process of creating a total ordering of unassigned components (in the current step of the planning algorithm) as described in section 5.5.2. This process represents one of the primary heuristics of the planning algorithm. The process is realized by component selectors, that are classes extending the `ComponentSelector` class. The `ComponentSelector` class is an `Iterable` over unassigned components and it defines a public interface and a simple framework for component selection:

- **newRun:** This abstract method must be implemented by the specific component selectors. It must create a total ordering of unassigned components and make it available in the `iterator` method defined on the `Iterable` interface. Its arguments are the results of component matching and the current planning state (i.e. `PlanningContext`). The planning context can be used to perform component selection which is based on the current partial planning result.

- **getName:** Returns the name of this component selector used for registration in a plugin registry.

- **init:** This method is called by the planning algorithm to initialize the component selector. It stores several parameters for use by the implementers and calls the `newRun` method.

We have provided several implementation of component selectors for demonstration purposes and comparison, with the best one of them being default:

- **Most restricted (default):** The default component selector orders the components by the number of their matching nodes in ascending order, thus preferring the more restricted or constrained components. This way the component with the lowest number of nodes that match its planning requirements is selected first. This trims the search tree of the algorithm early with the intention to prevent making bad planning decision too far in the future. It is one of the common heuristics for a depth-first search planning algorithm.

  There can be components with equal number of matching nodes. In this case this component selector prefers the component with the higher number of preferred nodes (intersected with the matching nodes) as specified in the node restrictions for the component. This sends the search of the planning algorithm in a direction which produces results more similar to the user's preferences.

- **Least restricted:** This component selector orders the components in reverse order than the default one.

- **Random:** This component selector orders the components in random order.

6.2.4 Node Selection

Node selection is the process of creating a total ordering of nodes that match the planning requirements of the currently selected unassigned component (in the current step of the planning algorithm) as described in section 5.5.3. This process represents another heuristic of the planning algorithm. The process is realized by node selectors, that are classes extending the `NodeSelector` class. The `NodeSelector` class is an `Iterable` over nodes and it defines a public interface and a simple framework for node selection:

- **newRun:** This abstract method must be implemented by the specific node selectors. It must create a total ordering of nodes and make it available in the `iterator` method defined on the `Iterable` interface. Its arguments are the nodes which match the planning requirements of the currently selected unassigned component and the current planning state (i.e. `PlanningContext`). The planning context can be used to perform node selection which is based on the current partial planning result.

- **getName:** Returns a name of this node selector used for registration in a plugin registry.
- **init:** This method is called by the planning algorithm to initialize the node selector. It stores several parameters for use by the implementors and calls the `newRun` method.

We have provided several basic implementations of node selectors. One of them which is usable most of the time is selected as default. The others are provided for demonstration purposes and they can be more useful for specific deployments. The provided node selectors are:

- **Prioritized random (default):** The default node selector is simple and fast, as it orders the nodes mostly randomly. However to create a planning which is similar to user's preferences it puts the preferred nodes at the start. For hybrid assemblies it prefers the node which is already selected for most of its subcomponents, thus planning the assembly close to its subcomponents.
- **Least used node:** This node selector orders the nodes by the number of components planned onto them, in ascending order. Thus it first selects the node which has the lowest number of components already planned onto it. This behavior enforces a very uniform distribution of components onto nodes.
- **Most used node:** This node selector orders in reverse order than the least used node selector. The resulting planning is very tightly grouped onto a few nodes and is very strongly influenced by the order of component selection.

6.3 Deployment Plan

Final part of planning is the generation of a deployment plan, if a valid assignment was found for all components and connections. The deployment plan is generated in two steps. The first steps creates a generic skeleton of the deployment plan void of component model specific metadata. The skeleton describes the structure of the planned component applications and stores the planning decisions. The second step modifies the deployment plan based on the requirements of the deployment runtime for various component models. The second step is another extension point of the planner as the modifications are performed by plugins (deployment plan metadata providers) specific for the component models.

6.3.1 Generic Deployment Plan

The `DeploymentPlanGenerator` creates the generic skeleton of the deployment plan. The deployment plan generator not only creates the plan, but also provides a basic framework for future modifications of the deployment plan by deployment plan metadata providers (via utility methods). This section provides an overview of the process which creates the generic deployment plan, whose class model is described in section 2.6.

The `ComponentInterfaceDescription` of the deployment plan is taken from the top-level assembly. Configuration property mappings represented by `PlanPropertyMapping` are also taken from the top-level assembly, which contains property mappings of the original component application merged during the flattening phase.

A `MonolithicDeploymentDescription` is automatically created for each successfully planned component (which can be a simple monolithic component or a hybrid assembly). The execution parameters and requirements of the monolithic deployment description are copied from the component metadata, i.e. from a `MonolithicImplementationDescription`. For each artifact of the component, an `ArtifactDeploymentDescription` is created. The artifact deployment description contains the location of the artifact, execution parameters and requirements copied from its description in the component metadata. The artifact deployment description also contains information about which resources were chosen to satisfy its requirements, and the name of the node where it is planned.

The instance of each successfully planned component is represented by one `InstanceDeploymentDescription`. The instance deployment description contains the name of the target node where the component will be instantiated, description of the resources which were chosen to satisfy its requirements and configuration properties copied from the component metadata. Each instance deployment description references a monolithic deployment description, which describes artifacts and execution environment needed to create the component instance.

Locality constraints in the component metadata can constrain the relative position of component instances in the domain (e.g. "must run on the same node"). These locality constraints are not stored in the deployment plan, because the planning algorithm automatically uses them to plan the component instances on correct nodes. But there are several types of locality constraints which describe the process separation of component instances on the same node (e.g. "must run in separate processes"). These are stored in the deployment plan via `PlanLocality`, and are processed by the deployment runtime.

A `PlanConnectionDescription` is created for all connections. They are connected to the correct ports of components and their requirements are

copied from the metadata of their respective connections. Each plan connection contains a description of deployed resources onto interconnects. Generally a connection can span multiple interconnects and bridges, thus it can allocate resources on multiple elements of the domain. However, this feature is currently unimplemented, thus a connection can be planned only on one interconnect.

6.3.2 Component Model Specific Metadata

The deployment plan must conform to the requirements defined by the deployment runtime. These requirements are specific to different component models, thus we use deployment plan metadata providers to modify the deployment plan in such a way which meets the requirements (see section 5.6). The metadata providers can modify the deployment plan in any way required by the deployment runtime for a specific component model.

The modifications of the deployment plan are performed by subclasses of `DeploymentPlanMetadataProvider`. The deployment plan metadata provider implements the `ComponentVisitor` interface, which presents the visitor pattern on the component metadata. Thus the component visitor defines `visit` methods which get called on all important classes of component metadata, i.e. `ComponentUsageDescription`, `ComponentImplementationDescription`, etc. An adapter on the component visitor is provided which defines an empty implementation of all `visit` methods, thus simplifying implementations of the visitor (so that even the `DeploymentPlanMetadataProvider` class extends the adapter). To sum it up, `DeploymentPlanMetadataProvider` presents an empty implementation of all of the `visit` methods and defines additional methods for its subclasses:

- **getName:** Abstract method which returns the name of the metadata provider, used for registration in a plugin registry.
- **preVisit:** Abstract method which is called before the visitor of component metadata is started. Can be used for initialization, pre-processing etc.
- **postVisit:** Abstract method called after the visitor of the component metadata finishes. Can be used for cleanup.
- **addImplementation:** Utility method adding a new monolithic deployment description to the deployment plan. It is used by metadata providers that add custom component instances to the deployment plan, as in case of Fractal.
- **addInstance:** Utility method adding a new instance deployment description to the deployment plan. It is used by metadata providers that

add custom component instances to the deployment plan, as in case of Fractal.

- **getImplementation:** Utility method which finds a monolithic deployment description for a specified component implementation. Used to modify monolithic deployment descriptions, e.g. by setting an execution parameter.
- **getInstance:** Utility method which finds an instance deployment description for a specified component. Used to modify the instance deployment description, e.g. by setting a configuration property.
- **getMergedConnection:** Utility method which finds a connection which was created by merging of a specified connection during the flattening phase of planning. This is used for example for Fractal in case of a virtual assembly, as is elaborated in later section 6.4.2.

The `DeploymentPlanMetadataProvider` subclasses have the planning results and a deployment plan (possibly already modified by other metadata providers) in disposal. They override some `visit` methods from the `ComponentVisitor` interface (thus overriding the empty implementations of the methods in the visitor adapter) to traverse the component metadata and use them to modify the deployment plan. The most common overridden method will be the one for visiting the component implementation description. The decision whether the deployment plan will be modified by the metadata provider and how will it be modified depends purely on the metadata provider. The metadata providers must detect which components are implemented in their respective component model with their custom logic. This will be usually done by detecting some special configuration property of the component (as is the case with Fractal). Thus the metadata providers impose additional requirements on the component metadata which are specific to the component model (i.e. they require a special identification configuration property). The planner itself does not impose additional requirements on the component metadata.

The deployment plan metadata providers are very generic, as they can perform arbitrary modifications of the resulting deployment plan. Thus they can be used not only to perform component model specific transformations of the deployment plan, but also to modify it to support additional technologies. The primary target for this feature is support of the connector framework, which can create connectors between components (see section 3.4). Metadata provider for connectors would traverse the deployment plan and convert all connections into a connector consisting of multiple connector units. The connector metadata provider would need to be run as the last metadata provider so that it would see all the modifications of the component model specific metadata providers.

6.4 Fractal

Fractal is the only component model currently supported by the deployment runtime, with requirements imposed on the deployment plan as described in section 3.3. Planner fulfills these requirements via the the Fractal metadata provider.

The component model of the OMG D&C specification supports only purely virtual assemblies which have no instances at runtime. This is in contrast with the Fractal component model. In section 3.2 we have proposed hybrid assemblies, which have a significant impact on the implementation of the Fractal metadata provider. The Fractal metadata provider supports both the unmodified component data model and the model with hybrid assemblies, so we can evaluate the impact of the change to the component data model.

In the following sections we examine the implementation of the Fractal metadata provider and its requirements on component metadata. The Fractal metadata provider supports both original component data model and the one enhanced with hybrid assemblies. The Fractal metadata provider does not require modifications of the platform independent component data model of the OMG D&C specification. It uses configuration properties of components to identify components implemented in Fractal and to store concepts unsupported by the component data model. The required configuration properties represent the requirements imposed by Fractal on the component metadata, but overall the Fractal metadata provider is fully compatible with the component data model of the OMG D&C specification.

6.4.1 Component Metadata Requirements

The Fractal metadata provider imposes requirements on the metadata of the component. Each component which is implemented in Fractal must be identified by a configuration property of the component implementation (stored in the `ComponentImplementationDescription` class). The property name is `component-model` and it must have a value of `fractal`. Please note that the name and value of the configuration property is defined by the Fractal metadata provider, not by the planner. Metadata providers for other component models can use different properties, or even completely different mechanisms to identify their respective components. All the following Fractal specific metadata are defined by the Fractal metadata provider.

The deployment runtime Fractal support requires the identification of the type of the assembly (see section 3.3). A Fractal assembly can be either `composite` for a purely virtual assembly or `hybrid` for an assembly with its own business code. This notion of virtual vs. hybrid assemblies is different than the one in the component data model, because Fractal's composite (i.e. virtual) assemblies are instantiated, and thus they need a description of

their artifacts. The Fractal's composite components do not have their own business logic (like serving web pages etc), even though they are instantiated. On the other hand they do provide some services which are Fractal specific (i.e. control of their subcomponents, lifecycle, etc.). The planner cannot automatically determine the type of the assembly (both composite and hybrid assemblies have artifacts) thus the type must be provided in the component metadata. This is done via the `fractal-assembly-type` configuration property of the component implementation. The value of the property can be `composite` or `hybrid`. The `composite` assembly type is assumed as default if the property is unspecified.

The planner does not generate connectors automatically (as described in section 3.4) yet. The connectors are manually stored in the component metadata. The connectors are implemented in Fractal, but they must be ignored by the Fractal metadata provider because they are not regular Fractal components taking are part in the component application. The connector components are be identified by the configuration property named `is-connector`, with any value.

6.4.2 Virtual Assemblies Implementation

Fractal's assemblies are instantiated at runtime, but the original component data model of the OMG D&C specification does not support such assemblies. The first approach used by the Fractal metadata provider to overcome this limitation is by requiring a special monolithic subcomponent of the virtual assembly, called `Fractal assembly implementation`. This monolithic implementation stores the information needed to create the instance of the assembly. The Fractal assembly implementation is not connected with other subcomponents, and no configuration properties of the assembly are propagated to it. It is basically isolated from the regular subcomponents of the assembly. The Fractal assembly implementation is handled specially by the Fractal metadata provider, and it must be identified by the `fractal-assembly-implementation` configuration property of the assembly. The value of the property is the name of the subcomponent which is the Fractal assembly implementation. A simple example of a Fractal assembly description is provided in figure 10. The dashed subcomponent M in the figure is the Fractal assembly implementation. It is not connected to other subcomponents of the assembly, and its name M is stored in the `fractal-assembly-implementation` configuration property of the assembly.

The information stored in the Fractal assembly implementation component is sufficient to create the instance of the assembly. But during the flattening phase of planning, the assembly was recursively removed and its connections merged (see section 5.3). Thus the automatically generated ba-

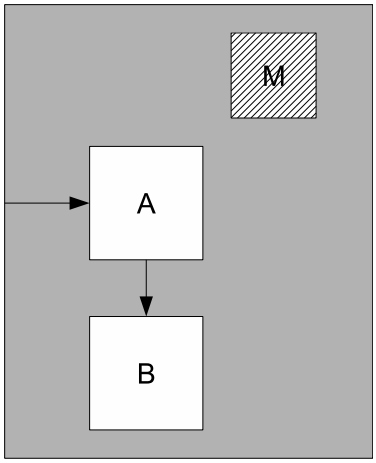


Figure 10: Fractal assembly implementation

sic deployment plan (see section 6.3.1) does not contain the descriptions of the assembly instance and its connections. The whole logical structure of the assembly must be re-created by the Fractal metadata provider. Fortunately he metadata provider has all the information needed to re-create the assembly structure (from the flattening phase). The original logical structure of the component application is traversed by the metadata provider as described in section 5.6. When the metadata provider visits an assembly, it uses the information stored in the Fractal assembly implementation subcomponent to create the description of the component instance in the deployment plan. The flattening process stores which connections were merged into one final connection, and this information is available to the metadata provider. With this information the metadata provider re-creates the descriptions of the assembly's connections in the deployment plan.

The instance of the Fractal assembly must be planned onto a specific node. The planning algorithm does not understand the special nature of the Fractal assembly implementation, and thus it can assign in on any node without regard to its subcomponents. The Fractal metadata provider can override this decision when it re-creates the instance of the assembly in the deployment plan. The policy which is used by the metadata provider to select a node for the assembly is selected via the `fractal-assembly-planning-policy` configuration property. Valid values are:

- `planner` - the node selected by the planning algorithm is retained. This is the default planning policy.
- `provider` - planning policy which is useful for assemblies which are mainly providers of services. The policy finds all subcomponents of the assembly to which a provider port of the assembly is delegated. Then, the policy chooses that node, which was already chosen by the planner

for the most of the above mentioned subcomponents.

- `user` - planning policy which is useful for assemblies which are mostly users of services provided by other assemblies. This policy finds all subcomponents of the assembly which have a user port delegated to a user port of the assembly. Then the policy chooses that node, which was already chosen by the planner for the most of the above mentioned subcomponents.

- `auto` - planning policy which combines the approaches of the provider and user planning policies. It tries to guess which one of the policies should be used. It chooses 2 nodes, one by the provider and one by the user policy. Then from these 2 nodes, it selects the one which was chosen for more subcomponents by the planner.

The deployment plan represents a flat component application without any notion of component hierarchy. Because of that the Fractal support in deployment runtime requires that the Fractal metadata provider stores information about the component hierarchy in the deployment plan. There are two ways to accomplish this, via special connections which represent the parent-child relationships between components, or by configuration properties which store the names of the parent and child components (see section 3.3). The Fractal metadata provider uses the configuration properties for their simplicity. For each assembly it automatically generates the configuration properties which contain the names of the instances of its subcomponents. For each subcomponent of the assembly, it creates a configuration property which contains the name of the assembly instance. These properties are used by the deployment runtime to automatically generate the special connections representing the parent-child relationship. However, using only properties to represent the parent-child relationship in the metadata provider has one drawback. If connector generation framework would be implemented in the planner, it would not be able to add connectors to the parent-child connections as these would be generated at runtime from the parent-child properties. This drawback could be overcome by using the connector generator in the runtime. Another alternative is to modify the Fractal metadata provider to generate the parent-child connections when the connector generator is implemented in the planner. This feature is a task for future work on our deployment framework.

The Fractal metadata provider modifies the interface of each assembly so that each of its ports has an internal copy as required by the deployment runtime Fractal support (see section 3.3). This is a simple task which also requires a minor modification of connections which connect the ports of the assembly with its subcomponents.

As can be seen this implementation of the Fractal metadata provider is

complex. The task of re-creating assemblies is complicated and duplicates some work done by the planner during the creation of the basic deployment plan. The metadata provider also duplicates the work of the planning algorithm, as it selects nodes for the assembly instances. All this shows us that the original component data model of the OMG D&C specification creates significant problems for component models which support instantiable assemblies.

6.4.3 Hybrid Assemblies Implementation

We have introduced hybrid assemblies to the component data model of the OMG D&C specification. These assemblies contain all information necessary to create their instance at runtime. The Fractal metadata provider supports the hybrid assemblies as an alternative approach to overcoming the limitations of the purely virtual assemblies of the OMG D&C specification. If the Fractal metadata provider encounters a hybrid assembly, then its work is greatly simplified compared to purely virtual assemblies (see section 6.4.2). The instances of the hybrid assemblies are automatically created and planned by the planner, together with their connections. Thus the metadata provider does not need to re-create anything in this case. Hybrid assemblies also need to store the component hierarchy information and their interfaces must be modified to contain internal ports. This is done very similarly as in case of virtual assemblies, and it is a simple task.

The job of the Fractal metadata provider is much more simpler in case of hybrid assemblies. The implementation code is several times shorter than the code to support the virtual assemblies. No work of the planner is duplicated in this case, and the planning algorithm has full knowledge of the assemblies and their planning. This demonstrates that the addition of hybrid assemblies to the OMG D&C specification has clear benefits for component models which support instances of assemblies.

6.5 Plugins

The planner defines several extension points which can be used to improve its support for various technologies, component models and additional heuristics. The extension points can be extended via plugins implementing required interfaces, as described in previous sections. The plugins must be registered and activated in a basic plugin registry implemented in the `PluginRegistry` class . The plugin registry maintains named collections of the different types of plugins:

- component matchers
- connection matchers

- component selectors
- node selectors
- deployment plan metadata providers

The plugin registry provides features for adding new plugins in runtime and configuring which ones are active. Currently the plugin registry is quite simple, without any advanced features such as automatic loading of plugins from JAR files etc. Its current main purpose is a centralized store of all plugins, including those that implement the core functionality. One the other hand complex features are not that necessary, as the planner is meant to be integrated within other tools, such as our GUI. The tools using the planner can implement complex plugin frameworks and use the plugin registry to extend the planner.

Runtime configuration of the plugin registry can also be used to select specific heuristics which are more suitable for the current planning problem. This way multiple heuristics can be provided with the planner and the user can choose the one which best suits his current needs.

6.6 Metadata storage

The OMG D&C specification describes repository interfaces which store the component data model and the target data model, as described in section 2. These repositories store the information required for the planning.

The models as defined by the OMG D&C specification do not have any business logic by themselves, i.e. they do not specify any operations on their classes. The models are intentionally defined in such way that it is possible to use them with the *Model Driven Architecture* (MDA) [22]. MDA is an approach to software design introduced and promoted by the OMG consortium. It is a software engineering paradigm in which software systems are described and developed by a series of models and model transformations. Actual software code is generated by automatic tools from the models. Generally MDA relies heavily on automated tools communicating with each other via the models. One of the goals of MDA is the separation of design and architecture of software systems. An important aspect of MDA are model transformations, which can transform general platform independent models into platform specific model (e.g. for a specific technology). Models of the OMG D&C specification are platform independent and they are intended to be transformed into platform specific models for specific component models. Our work does not create the platform specific models, but uses the facilities provided by the platform independent model.

As the models are intended to be used with MDA, we were able to describe the models in a high level language and use automatic tools to generate code

which represents the models. We have used the *Java Architecture for XML Binding* (JAXB) [23] framework to describe the models via an XML Schema and generate Java code from such representation. The JAXB-generated code automatically handles serialization and deserialization of instances of the model classes to and from XML.

Alternative MDA relevant tools are MOF [27] and EMF [29]. MOF was originally used for this work, but was abandoned in favor of JAXB as its development was stagnating. EMF is a much newer development and was considered in the later stages of this work. However, we continue to use JAXB as it provides us with all the features necessary at this stage.

Model driven development with JAXB greatly simplifies creating the code of the model classes and the repositories. The model classes are described in a simple XML Schema and automatically transformed into Java code by an automatic tool. The XML Schemes are a higher level language for describing the models and generally it is much easier to define the models in the schemes than directly via Java code. Changes to the model are simply done by modifying the XML Schemes and re-generating the code, thus simplifying development and maintenance. The repositories only add the business logic to the storage of the model (e.g. caching of metadata, checking for installation under an already used name etc), but the serialization and deserialization code is automatically generated by JAXB.

A repository for managing the target data model was implemented in [7]. An existing component model repository was improved as part of this work. Both repositories use the code automatically generated by JAXB to represent and store the models. The component repository implements the `RepositoryManager` interface of the component management model (see section 2.3). This interface handles the storage of component metadata, but not the storage of artifacts. The component data model assumes that artifact data is stored in arbitrary locations, usually represented by a URL. This way the artifact data can be available from a HTTP server, network filesystem etc. The artifact location must be supported by the specific deployment runtime. Our deployment runtime originally supported only artifacts stored in locally accessible files or via an HTTP server. The component repository was enhanced to store the artifact data internally to remove the need to use a separate HTTP server for remote access to artifact data. The deployment runtime was extended to support the download of artifact data from the component repository.

6.7 Examples

Several example component applications are provided with this work. Demo logging applications implemented in the Fractal component model test the integration with the deployment runtime, as these component applications

are instantiable and can be executed. The Fractal logging applications are re-used from the deployment runtime implementation [7] by creating their metadata for the component repository. A more complex component application represents a simple typical information system. The information system application cannot be run by the deployment runtime, as only its metadata without an actual implementation are provided. The information system application is designed to test more complex planning tasks, as its logical structure is more complicated than the Fractal logging demo.

6.7.1 Local Fractal Demo

The local Fractal demo represents a simple flat component application able to run only on one node. Its basic structure is depicted in figure 11. Three types of components implement the demo application - Message Producer, Message Arbitrator and Log Factory. The Message Arbitrator uses the Log Factories to create logs, which are then used by the Message Producer to log messages.

The Message Arbitrator communicates with an arbitrary number of Log Factories. It randomly selects one and creates a log via its business interface. The log is then passed to the Message Producer and is set as its current log. The Message Producer generates log messages containing an increasing index and logs them via the log received from the Message Arbitrator. This causes the Log Factories to output the log message to the standard output. Thus the output of this demo application are log messages generated by the Message Producer and printed to standard output. The output also contains information messages notifying of the change of the current log of the Message Producer.

Even though this component application is flat, it must be encapsulated by a top-level assembly named Demo which represents the whole application in the component repository. The assembly is purely virtual and it is not instantiated in runtime. The planner removes the assembly during the flattening phase.

This demo application tests basic integration of the planner with the deployment runtime and its ability to run simple component applications. The metadata of the component application contain information specific for the Fractal plugin of the planner, which are used by the Fractal planner plugin to generate information required by the deployment runtime in the deployment plan.

6.7.2 Hierarchical Fractal Demo

The hierarchical Fractal demo runs business code very similar to the local demo, i.e. the Message Producer logs messages via a log created by the Log Factory chosen by the Message Arbitrator. However the application is

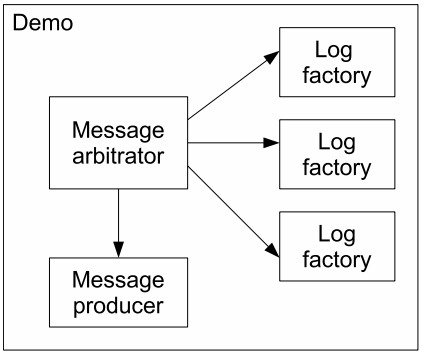


Figure 11: Fractal local demo

hierarchical as can be seen on figure 12. The part of the application that handles the logging is separated into the Logger assembly. This assembly hides the Message Arbitrator and Log Factories from the Message Producer, which communicates only with the Logger. The connection between the Logger and the Message Producer is delegated onto the Message Arbitrator.

The top-level Demo assembly is also instantiated. Its business code prints the hierarchy of the application onto standard output, thus demonstrating the ability of hybrid Fractal assemblies to contain business code and access the component hierarchy information.

As both assemblies of this application are instantiable, they must contain the information necessary to create the component instance as we have discussed in section 6.4. The Fractal planner plugin supports two ways of storing the required information, using either a special monolithic subcomponent, or by the use of a modified OMG D&C specification data model which adds hybrid assemblies. This demo application demonstrates the use of a special monolithic subcomponent, a *Fractal assembly implementation*. The subcomponent is highlighted in gray in the figure. As can be seen, this approach puts additional requirements on the packager of the component application. The resulting component metadata has a slightly different hierarchy than the logical hierarchy of the appliation, which can be confusing. On the other hand this approach does not require any modification of the OMG D&C specification.

The hierarchical Fractal demo can be executed in a distributed environment. The connections between the components use connectors to handle the network communication. The planner is not integrated with the connector framework yet, thus the connectors are not automatically generated and they must be manually provided in the component metadata. Each connection in the component metadata is represented by three connections and two connectors as can be seen in figure 13. One connection connects the client component with the client connector unit. The client component and the client connector unit will be run on the same node thus their communication

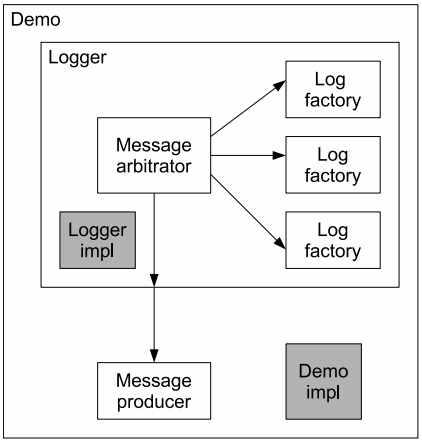


Figure 12: Fractal hierarchical demo

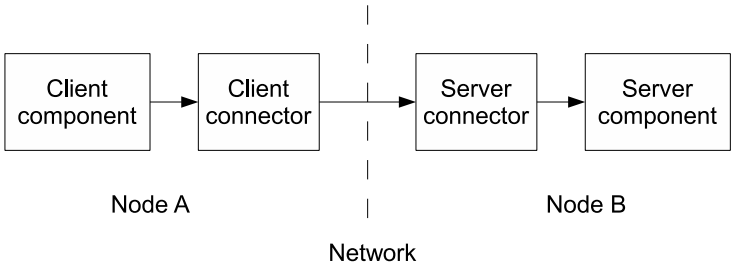


Figure 13: Connectors

is local. The client connector facilitates the communication with the (possibly) remote server connector unit over the network. The server connector unit and the server component will be also run on the same node and thus communicate locally. The planning of connector units on the same node as their respective components is accomplished via locality constraints.

As the planner is not integrated with the connector framework, the manual insertion of connectors into the component metadata requires additional work from the packager of the component. The resulting component metadata have a more complicated logical structure than without the connectors. Thus the integration of the connector framework with the planner will bring clear benefits.

6.7.3 Hybrid Fractal Demo

The hybrid Fractal demo demonstrates the usage of hybrid assemblies which were proposed in section 3.2. Otherwise the application is the same as the hierarchical Fractal demo with the same business code and output, of course without the Fractal assembly implementation components.

This demo application demonstrates the benefits of the hybrid assemblies.

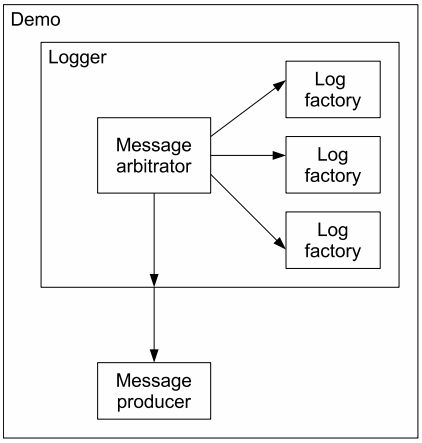


Figure 14: Fractal hybrid demo

The metadata of the application have the same logical structure as the application, without any confusing special monolithic components. Packaging such an assembly is more straightforward. The result of the planner contains the assemblies thus the Fractal plugin does not need to re-create them.

Two additional versions of the hybrid Fractal demo are provided for performance testing. They have the same logical structure, but contain 50 log factories together with their respective connectors, thus the total component count is around 150. One of the larger demos has basic requirements which do not require any allocation of resources. The other demo specifies memory size requirements for the log factories, which force the planner to spread the log factories in the domain in such a way which does not over-allocate the available memory on any of the hosts.

6.7.4 Information System

We provided the metadata of a component application with a structure similar to a simple business information system, as shown in figure 15. The aim of this component application is to test the planning algorithm, but not the integration with the deployment runtime. We only provided the metadata of the application, not an actual implementation. The metadata of this application do not contain any artifacts. As there is no implementation, the application cannot be run by the deployment runtime and contains no component model specific metadata.

The application is separated into Backend and Backup assemblies, with Backend implementing the core functionality and Backup storing backups and notifying the Backend that new backups should be created. The Backend assembly consists of a Log Storage component and a Business Logic assembly. The Log Storage is used by the Business Logic component to store its logs which are further sent to the Backup component for backup. The Business

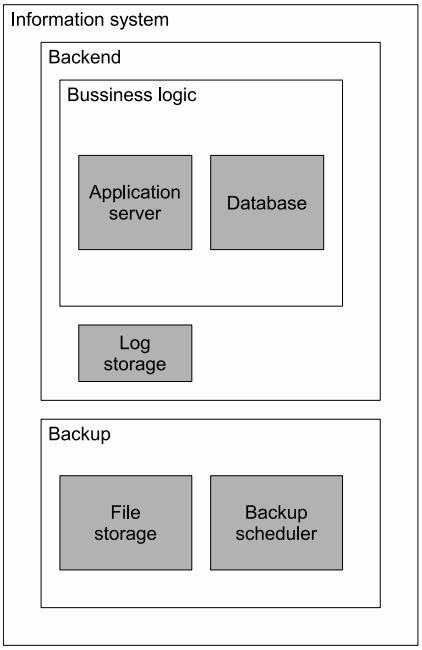


Figure 15: Information system

Logic assembly contains the two main components of the application, an Application Server (e.g. JBoss) which serves a web user interface to the user and communicates with a Database. Both the Application Server and the Database create backups via the Backup component.

The whole component application contains a wide range of connections between its components, which are not shown in the figure for clarity. The components and connections impose requirements on the resources available in the domain. Locality constraints force the planner to create a deployment plan which is more performance aware. Some components can be configured via configuration properties which are propagated from the top-level assembly.

All assemblies of this application are purely virtual, thus they are removed during planning and merged with their subcomponents. This demonstrates the use of virtual assemblies as means of component composition.

6.8 Tools

Tools to ease the further development of the planner are provided with this work. Their basic idea is running and testing of the planner without the need to recompile it. As the planner is extensible via plugins which can modify and enhance its behavior, the plugins are easily configurable.

Two command line utilities serve the testing and development needs of the planner. The *Planner tester* runs the planner with a specified configuration

and generates a deployment plan in an XML file. The deployment plan is then simply run by the *Plan runner* in the deployment runtime so that the component application can be observed in the deployment runtime.

Both utilities are started via an Ant script [24]. The benefit of the Ant script is its simplicity and inherent multiplaformness. The Planner tester is run via the `test-planner` target of the Ant script, i.e. by the command `ant test-planner`. The Plan runner is run via the `run-plan` target of the Ant script, i.e. by the command `ant run-plan`.

Configuration of both tools is read from standard Java property files. The property files used by default are `planner.properties`, `log.properties` and `node-restrictions.properties`. The locations and names of these property files can be changed on the command line when launching the Ant target.

The `planner.properties` file controls the main behavior of the Planner Tester and Runner. It specifies the filename of the generated deployment plan file, with `deployment-plan.xml` as default. Installation name of the component application which will be planned is specified in the `package.name` property. The plugins which implement the planner's behavior and heuristics (see section 5.5) are specified in the `component.matchers`, `connection.matchers`, `component.selector`, `node.selector` properties. Default plugins are used if the properties are omitted. Used metadata providers (see section 5.6) can be specified via the `metadata.provider` property, with default ones being used if the property is omitted. The Planner Runner runs the deployment plan via a possibly remote Execution Manager, with its host specified in the `execution.manager.host` and `execution.manager.port` properties.

The planner uses the Log4j [25] logging framework to print progress and debug information. The logging can be fine-tuned via the `log.properties` file. Several loggers can be configured separately - the printing of input metadata of the component and domain, matching of planning requirements, core planning algorithm progress and logging of the planning tools. These loggers can be independently configured to print more detailed information specific to the current needs of the user or developer.

Node restrictions of the various components (see section 5.2) can be specified via the `node-restrictions.properties` file. The file contains name-value pairs, with the name being a path to a component and value being the component's node restrictions. The path to the component is the path over the component hierarchy, i.e. it contains the names of the component implementations and the subcomponent names. For example `/fractal_demo/logger/logger` is a valid path, with `fractal_demo` being the name of the implementation of the top-level assembly, `logger` is the name of a subcomponent of the assembly and the final `logger` is the name of the subcomponent implementation (see section 2.2 for a description of multiple component implementations). The node restrictions for the component

are specified as space separated node names, with possible prefixes. No prefix or a `+` sign means that the node is allowed. A `-` sign as a prefix means that the node is forbidden, and the `!` prefix specifies a preferred node. For example, `-Node2 !Node1` means that the node Node1 is preferred and the node Node2 is forbidden.

The property files provide a simple means of configuration of the planner tools. They are persistent (unlike command line options which must be entered each time again), multiple configurations can be represented by multiple copies of the files, and they can strongly modify the behavior and output of the planner. The Planner Tester and Runner tools provide simple means for testing and debugging the planner without the need to use a GUI.

7 Planner GUI

Planning of component applications is a highly interactive task even with the help of an automated planning tool. The end users want to influence the resulting planning because of their subjective preferences, or because they have better knowledge of the deployment environment than the automated tools. The planner and the user need to "communicate", so that the user can easily see the generated planning and modify it. The process of creating a planning should be interruptible so that the user can store a partial result and finish it later. The resulting planning needs to be persisted for possible later re-use and the user needs a way to execute the deployment plan.

The process of planning requires additional information to be presented to the user. The user needs to be able to examine the target domain, its structure and resources provided by its elements. Planned component applications need to be examined to overview their structure, requirements, configuration properties and binary artifacts.

This work focuses on the planning phase of the deployment process as specified in the OMG D&C specification. A graphical user interface (GUI) is supplied with the work which implements the above mentioned use cases. The GUI focuses primarily on providing a user interface for the planning phase. However this is just a part of a GUI which would be needed to interact with the complete deployment framework. Creating such a complex GUI is out of scope of this work, but it would be beneficial if it would be possible to enhance the GUI in the future with additional features.

The planning GUI aims to accomplish several goals:

- **Computer assisted planning:** The user will be provided with a user-friendly planning GUI. The GUI will automatically assist the user in creating a valid planning with the help of the planner engine.
- **Component view:** Metadata of components will be presented to the user in a form suitable for hierarchical components.
- **Domain view:** A view of the domain will present the information about the target domain, its elements, structure and resources.
- **Extensibility:** It will be possible to extend the GUI in the future to support other parts of the deployment framework and the deployment process.

7.1 Eclipse Platform

The planning GUI is based on the Eclipse platform [12] [13]. The Eclipse platform was chosen for its multiplatformness, ease of development and extensibility. The Eclipse platform is designed to provide a good, extensible

and user-friendly platform for creating tools. The tools are primarily development tools like IDEs, but the platform is well suited for a wide range of graphical applications.

The fact that Eclipse is implemented in Java gives it great portability. The user interface is created in the Standard Widget Toolkit (SWT) [26]. Because of this Eclipse does not use the Swing toolkit which is a part of the Java environment. SWT is bundled with Eclipse so it is not installed separately. The main idea of SWT is the reuse of native widgets as much as possible. It is basically a thin wrapper around native widget toolkits presenting a unified API to its clients. SWT supports a variety of toolkits on multiple platforms, but it is not inherently multiplatform. It can be run only in an environment containing a supported toolkit. However the most commonly used toolkits are supported by SWT, so users of a big variety of platforms can use it. The use of native widgets has several implications. The main disadvantage is the "lowest-common-denominator" effect, which means that SWT design and features are limited by the least advanced of the supported toolkits. However there are significant benefits in the form of performance and most importantly native look-and-feel of Eclipse based applications. Some advanced widgets are not natively supported in all platforms, thus SWT implements its own set of advanced widgets. This way the SWT toolkit is a combination of a purely native widget toolkit and a custom Java based toolkit. More advanced features (as a *model-view-controller framework*) are provided by the JFace framework. SWT and JFace are integral parts of the Eclipse platform and all tools built upon it use them. On the other hand they can be used separately in Java applications which are not based on Eclipse.

The Eclipse platform is highly extensible by design. A large ecosystem of tools built upon it exists. The platform is designed in such way that the tools can cooperate and live side-by-side thus enhancing their functionality (e.g. Java development tools and tools providing integration with source control systems like CVS). This extensibility allows the planning GUI to be easily enhanced to support the future features of the deployment framework, and use other existing tools at the same time (e.g. Java integration). Multiple tools relevant to model driven development exist for the Eclipse platform, and could be used in the future to extend our GUI [28] [29] [30].

The Eclipse platform and the ecosystem of tools built upon it promote cooperation of the tools and an open source development model. The platform and a high number of tools built upon it are royalty-free and available under an open source license (the Eclipse Public License).

7.1.1 Extensibility

The whole Eclipse platform is built as a set of plugins. The core of the platform is a plugin loader, which facilitates the loading of plugin classes, resolving their dependencies and enforcing security and isolation. The plugin architecture is an implementation of the OSGi framework R4.0 specification [14]. The OSGi specification describes a framework for defining, composing a executing of bundles. The OSGi bundles are similar to components, and we can think of OSGi as a component model for Java. Plugins are the historically used name for bundles, and we will use the term plugin in this work.

The OSGi plugins are self describing. They describe their classpath and plugin dependencies and code exported to clients. Complex tools are created by combining multiple plugins using each other's services. OSGi plugins are separated by their classloaders, thus they are isolated from each other. Each plugin gets its own classloader, and the classpath is created based on the plugin's dependencies. The description of an OSGi plugin is provided in a `MANIFEST.MF` file.

The Eclipse platform adds another layer on top of OSGi. An extension registry provides a mechanism for defining another type of relationships between plugins. Plugins can specify that they can be extended or configured by other plugins by defining an *extension point*. The extension point declares that the plugin will do a specific functionality, if provided with some required information by client plugins. The client plugins provide the required information in the form of an *extension*. So for example, one of the main GUI plugins has a "New wizard" extension point declared. The extension point basically says that if another plugin provides the required information, then the main plugin will automatically show its wizard in the standard "File/New" menu. The required information is the name of a class which implements a wizard for creating new files or resource (which must extend a specific class), a name of the wizard, an icon and a few more configuration options. Each plugin can define multiple custom extension point, and at the same time contribute multiple extensions to other plugins. This is the principal mechanism used to create complex tools with the Eclipse platform. The extension points and extensions of plugins are defined in their `plugin.xml` files.

All plugins in Eclipse are versioned. The versioning information can be used in the specification of plugin dependencies (e.g. "I require plugin foo of the 1.5 version or higher"). Eclipse provides a centralized mechanism to update plugins to newer versions and to install new plugins.

Initially the Eclipse platform was intended for the development of IDE-like tools, e.g. for Java. But the platform proved successful and good enough that the need to create more generic rich client application arose. Eclipse was slightly modified to remove its "IDE-ness", and the Eclipse Rich Client

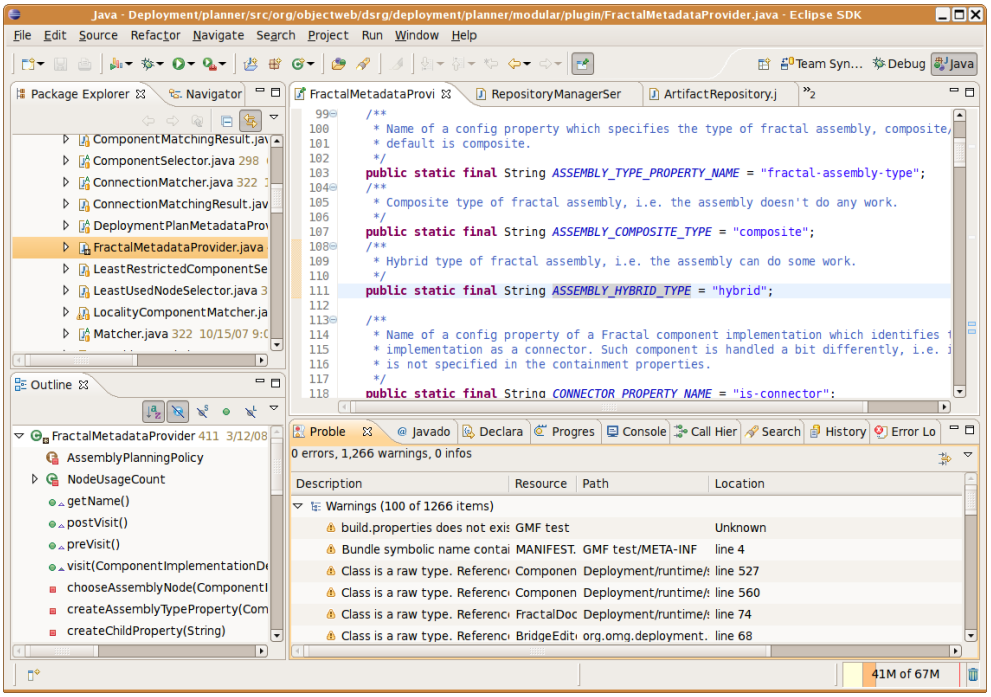


Figure 16: Eclipse workbench

Platform (Eclipse RCP) was created. The Eclipse RCP platform is a basic set of plugins which provide core functionality of the plugin system and basic user interface elements and tools. The full-fledged Eclipse IDE is now built upon Eclipse RCP, adding the IDE specific functionality.

7.1.2 User Interface Paradigm

Most Eclipse applications follow a consistent UI paradigm. The main element of their user interface is the workbench, which contains multiple *editors* and *views* with a menu at the top and a status line at the bottom. The editors and views can be moved and resized within the workbench with a high degree of freedom. However the layout of editors and views is limited in a way, by requiring the editors to be placed generally in the center with the views surrounding them. A sample screenshot of an Eclipse workbench is provided in figure 16. The screenshot shows a workbench for editing Java code. Source code editors are placed in the center. The editors are surrounded with views for browsing the project files and providing context information for the current editor (e.g. code structure outline).

Editors modify a resource (e.g. file) or any entity in general in the usual open-edit-save workflow. An Eclipse application can contain multiple types of editors for specific types of resources. So for example a Java source file is opened with an editor which understands the structure of Java source and

implements syntax highlighting, auto-complete etc. A different editor would open XML files etc. The user can choose which editor will be opened for a resource, and editors can be automatically associated with resource types (e.g. for files with the .txt extension). Editors occupy the central part of the workbench. Multiple editors can be open at the same time, and several of them can even modify the same resource at the same time.

Views usually browse or navigate resources and edit their properties immediately without explicit saving. They often provide contextual information relevant for the currently opened editor. Only one view of each type can be open at one time.

A *perspective* is a specific selection and layout of views and editors tailored for a task. For example different editors and views are appropriate for Java programming than for synchronizing with a CVS repository. Perspectives can be defined by the tools built upon Eclipse and modified by the users.

7.1.3 Planning GUI Integration

The planning GUI is based upon the Eclipse platform and it follows the usual Eclipse user interface rules. We have provided views for examining the domain and the component repository. Custom editors are supplied for viewing component metadata, domain element properties and for creating a planning. All these user interface elements create the *Deployment perspective*. The generated deployment plans are saved as files and can be later examined and modified with a custom editor. Deployment plans can also be executed and stopped.

The planning GUI is a plugin which depends on the Eclipse Java IDE, not just on the basic Eclipse RCP platform. The heavier dependency is for simplicity and ease of development of the plugin, and can be later reduced. The GUI plugin is packaged as one JAR file and is installed by simple copying into the plugin directory of Eclipse.

The views of the component metadata and of the domain are read-only. Providing a user interface with full read-write access to the model supporting all the usual GUI features as undo & redo, drag & drop etc. is out of scope of this work. Read only access is sufficient for planning, and the GUI can be extended in the future for read write access with the help of modeling tools as EMF [28] [29] and GMF [30].

7.2 Metadata Views

The planning GUI provides views and editors of the metadata used during planning. These views can be used by the user to examine the metadata of components and the domain and use the information during planning.

7.2.1 Target Manager

Examining the target domain is done via a *Target Manager view*. The view serves the purpose of examining the structure of the domain and properties of its elements. The view is located in the bottom left corner of the workbench by default. It presents a list of all nodes, interconnects and bridges in the target manager. The list is taken from the Target Manager, which must be up and running. The target manager view tries to connect to a Target Manager running on the localhost by default. If the Target Manager runs on a remote host, the user must enter the hostname and port in the view. This way the view does not require configuration during common local development.

Double clicking on any element of the domain opens an editor and brings it forward. The editor presents a read-only view of the domain element, containing a list of other domain elements connected to the viewed one and the resources and shared resources present on the element. So for example, opening an editor for an interconnect shows the interconnect's resources and a list of nodes connected to it. Double click on any connected element opens an editor for it, so the user can quickly traverse the domain.

7.2.2 Repository Manager

A list of component applications is available in the *Repository Manager view*. The view serves the purpose of examining the contents of the component repository and the logical structure and metadata of installed component applications. This view is located in the bottom left part of the workbench by default. It presents a list of all installation names of component applications in the component repository (i.e. the Repository Manager). The component view tries to connect to a component repository running on the localhost by default. If the component repository runs on a remote host, the user must enter the hostname and port in the view, similarly to the Target Manager view.

Double clicking on a name of a component opens its read-only editor. The editor is separated in two parts. The left side contains a tree of the logical structure of the component application. The nodes of the tree are components and artifacts. So for each component, its subcomponents and artifacts are shown as child-nodes. The tree view serves to quickly display the logical structure of the component application. The right side displays the metadata of the element selected in the left side. Selecting a component displays its complete metadata, as configuration properties, interface, requirements etc. Assemblies display their connections, locality constraints and property mappings. Selecting an artifact displays its locations, requirements and properties.

7.3 Planning

The GUI focuses on the planning of component applications. The planning task is potentially complicated and long lasting, with the user requiring assistance from the automated planner. The planning GUI solves three main aspects of the planning task - computer assisted planning, interruptibility and persistence of the deployment plan.

The planning task is represented via editing and creating a *deployment*. Deployment is a selection of a component application, a description of where the user wants to run its components and the configuration of the application. More specifically it is represented by the installation name of the application, a set of node restrictions (see section 5.2) created by the user and a set of configuration properties. Deployment is stored in a file (with a `.dpl` extension) and is edited by the user in a special editor. The information stored in the deployment is used by the planner to create a deployment plan. A new deployment is created either via new wizard available in the standard location in the menubar (File/New/Deployment), or by right clicking on a component in the repository manager view. A new deployment is empty, i.e. it contains no node restrictions and configuration properties. When the node restrictions or configuration properties are modified, the deployment must be saved in the standard way an editor is saved. Thus a deployment is persisted, and the user can return to it later and continue modifying it.

The deployment editor runs the automated planner *reactively*, i.e. on every change made by the user. When a new deployment is created, the planner is automatically run to try and create a planning without the user even having to do anything. The resulting planning might not suit the user or it may not be found. The user can modify the deployment, and immediately after each modification the planner is run again and the user sees its results immediately. This way the user "communicates" with the planner and gradually modifies the planning with the help of the automated planner until it suits his needs. We can think of such interaction as *computer assisted planning*.

When the desired planning is created, the deployment plan can be saved into a separate file (with a `.pln` extension). Thus the deployment plan is persisted and can be reused multiple times and edited in the future. The deployment plan is an XML file and we have provided a special editor for it. The editor shows the raw XML content in one tab, and a simplified view of the component instances in a separate tab. The instance view enables the user to quickly review or modify the nodes where will the instances run. The deployment plan can be executed in a standard Eclipse way via right clicking on its file, and selecting Run as/Deployment Plan. This action executes the deployment plan in the Execution Manager and shows an entry in the standard Eclipse progress view where the application can be stopped.

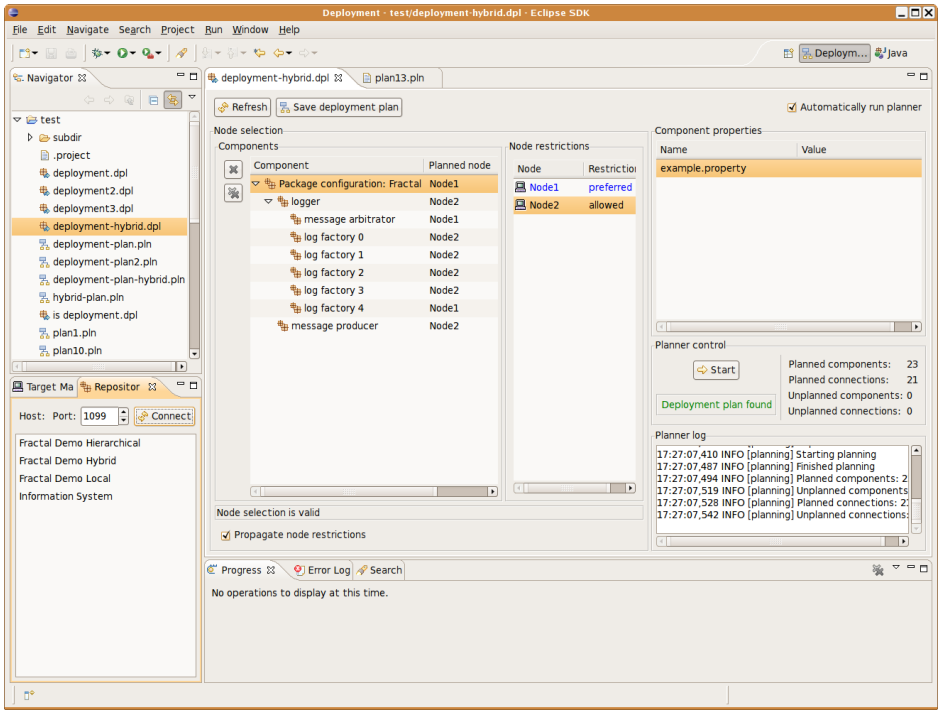


Figure 17: Deployment editor

The deployment editor is shown in a screenshot provided in the figure 17. The primary part of the editor is used for node selection. It displays a 2 column tree representing the logical structure of the component application, with the name of each component in the first column and the name of the node automatically selected for the component in the second column. To the right of the tree there is an editor of node restrictions for the component currently selected in the tree. For all nodes of the domain a restriction can be selected as one of `allowed`, `forbidden` or `preferred`. The default node restriction is set to `allowed`. The different types of node restrictions are differentiated by colors too. In the bottom left of the deployment editor there is a checkbox which enables propagation of node restrictions (enabled by default). If node restrictions are being propagated, then modifying a node restriction on a component propagates all of its node restrictions to its subcomponents. This is used as a quick way to set the node restrictions to a subtree of the component hierarchy.

The top right part of the deployment editor allows the editing of configuration properties of the component application. It display the properties specified in the interface of the top-level component. The properties are edited directly by clicking on their value.

The bottom right part holds a control interface for the automatic planner. The planner is launched automatically for every change in the deployment,

but in some cases it can take a long time for it to finish. The planner control interface allows the user to monitor the progress of the planner, stop it and review its logs. Planner is started automatically only when the checkbox in the top-right corner of the deployment editor is enabled.

When the user is satisfied with the planning, he saves it by clicking on the **Save deployment plan** button in the top part of the editor. He must specify a filename of the deployment plan XML file. The **Refresh** button is used to update the component metadata from the component repository, e.g. when the user modifies it while creating the deployment because he was unable to create a valid planning.

8 Evaluation

8.1 Goals

We have analyzed and implemented tools for the planning phase of the deployment workflow as specified by the OMG D&C specification. The tools are automated and support the deployment of heterogeneous component applications which are composed of components implemented in different component models. The planning tools consist of two parts - planner and a graphical user interface.

8.1.1 User View

- **Graphical interface:** We have implemented a graphical user interface based on the Eclipse platform and focused on the planning part of the deployment workflow defined by the OMG D&C specification. The user interface provides the user with all information needed during planning via specific views of the component metadata and the computer domain. The views show all information present in the data models of the OMG D&C specification in a clear form. Currently the views are read only because creating a complex editor which supports modification of the models with usual UI metaphors as undo & redo, copy & paste or drag & drop, is out of scope of this work.

- **Computer assisted planning:** The main focus of the planning GUI is the planning of component applications. The GUI interactively assists the user in creating a valid planning which suits his needs. A planning algorithm is integrated with the GUI. The algorithm respects the preferences of the user and searches for a valid planning automatically. The search is accomplished by backtracking controlled by several heuristics. Validity of the planning is checked from many aspects required by the OMG D&C specification, as resource requirements, component application structure etc.

  The planning algorithm is launched reactively to every action of the user, thus assisting him with gradually creating a planning which suits his needs and is valid at the same time. This way a new valid planning is generated by the GUI each time the user modifies his preferences. The resulting planning can be easily reviewed and modified. The whole process is interruptible to support the creation of more complicated plannings. Resulting plannings are persisted as files, and can be modified and reused later.

  The planning algorithm is fast enough in the common situations so that it can be used in highly interactive user interactions, e.g. planning of

150 components. Solving and benchmarking very complex plannings is beyond the scope of this work.

- **Extensibility:** As the planning GUI is built on the Eclipse platform it can be easily extended to support interaction with the whole deployment framework. The extension mechanisms of the Eclipse platform allow seamless integration of multiple independent tools. It is not be necessary to modify the existing planning GUI to integrate it with other tools.

8.1.2 Developer View

- **Extensible planning:** The planning algorithm has several key decision points which use heuristics to control the flow of the algorithm. The decision points control the flow of the algorithm (component and node selectors) and validate the planning (matchers). These decision points are easily extensible by extension of classes and registration in a plugin registry. This extensibility of the algorithm means that it can be extended with more advanced planning heuristics and with support of additional technologies such as connectors.

- **Heterogeneous component applications:** The deployment plan must contain component model specific metadata required by the deployment runtime. We have implemented an extensible mechanism represented by deployment plan metadata providers, which automatically adds the metadata into the generated deployment plan. The mechanism is general enough to be useful for additional technologies as connectors. Currently the planner supports applications implemented in the Fractal component model, as it is the only component model supported by the deployment runtime. We have analyzed the requirements of the SOFA component model.

  Heterogeneous component applications are described by the component data model of the OMG D&C specification. We use the facilities of the component data model (such as configuration properties) to describe aspects of heterogeneous component applications that are not directly supported by the component data model.

- **Integration:** Our planning tools are integrated with a deployment framework consisting of multiple parts. We use repositories which store the metadata of components and of the computer domain. The created deployment plan can be executed in the deployment runtime, as proven by demo Fractal applications demonstrating a logging system.

8.2 OMG D&C Specification

The component data model can be used to store metadata of heterogeneous component applications in its platform independent form. We do not create platform specific models for each component model, but we store all component model specific metadata via the facilities (configuration properties) provided in the component data model, as demonstrated in the example Fractal component applications.

However the component data model does complicate the metadata and deployment when using component models which support instantiable component assemblies, such as Fractal. Such assemblies need to store the information needed to create their instances. This is not directly possible in the component data model of the OMG D&C specification. On the example Fractal application we have shown that this limitation can be overcome by requiring a special monolithic subcomponent of the assembly which contains the instance information. This approach has several drawbacks such as more complicated component metadata and highly complex implementation. To overcome this problem we have proposed a slight modification of the component data model introducing hybrid assemblies. Hybrid assemblies allow a component implementation to have both an assembly of subcomponents and an instantiable monolithic implementation. This slight modification significantly simplifies the implementation of Fractal support in the planner, which supports both the original and the modified component data model. We anticipate that possible future implementation of additional component models would greatly benefit from this modification.

The execution data model of the OMG D&C specification represents a flat component application without any notion of component hierarchy. This is in contrast with several component models which require component hierarchy in runtime. The Fractal support in the deployment runtime utilizes configuration properties of component instances or special connections to store the parent-child relationships of components. Such approach can be used for other component models. Another possibility would be enhancing the execution data model with explicit support of the component hierarchy.

8.3 Future Work

The OMG D&C specification is quite complex and supports various advanced concepts and features. The planner supports most of the features omitting a few which implement less common use cases.

Heterogeneous component applications in the terms of the specification have a different meaning than the one used throughout this work. The specification's term means components with multiple alternative implementations, usually usable on a specific platform. One of the implementations must be

chosen during planning based on its requirements and capabilities. Such component applications greatly complicate their packaging and composing, and such a use case is really uncommon. Another problem is that the structure of such component applications is essentially two dimensional. This greatly complicates their use in any user-friendly way, especially during planning. The planning tools currently do not support multiple implementations of components.

The OMG D&C specification allows for components to serve as resources on the computer hosts. These resources can be used to satisfy the requirements of components. Implementing this feature requires enhancement of the deployment runtime as well as using the extension mechanisms of the planner to support such resources.

The target domain defined by the specification can contain multiple computer networks connected by bridges (i.e. routers). The executed component application and especially its connections can span multiple networks. The planner currently does not support planning connections on multiple networks. The algorithm would need to be enhanced to find paths in such complicated domains and plan connections on them. On the other hand such feature would be rarely used. In most cases the interconnected physical networks can be represented as one logical network because the resources provided by them are rarely significantly different from the perspective of deployment of common component applications.

The planner was designed for easy extensibility. The primary target for this was integration with the connector framework which would further enhance and simplify the deployment of heterogeneous component applications. This integration requires minor changes of the deployment runtime and a two step extension of the planner. The first step would determine whether it is even possible to create connectors between specific components (via a connection matcher, see section 5.5.1). The final step would modify the deployment plan by inserting the connector components. This step would use the same mechanism which is used to generate component model specific metadata (see section 5.6) as the mechanism is very generic.

Deployment of more complex and heterogeneous applications requires support of additional component models. One such candidate is the SOFA2 component model. Adding support for an additional component model requires extension of both the planner and the deployment runtime, as currently they only support Fractal.

The extension mechanisms provided by the planner can be used to improve it with more advanced heuristics. Currently the planner is not aware of the performance of executed component applications. Such information could be used for performance aware planning which would control and monitor the load of the computer nodes in the domain. There are techniques [18] for the collection of performance information from running component appli-

cations, which would have to be stored and made accessible for the planner.

The GUI can be extended to support the whole deployment framework and its workflow. As it is based on the Eclipse platform it can be seamlessly integrated with additional tools relevant to the deployment of component applications.

8.4 Related Work

There are two classes of related work. Deployment frameworks present and define the general concepts and technologies which are relevant to planning of component applications. Planning tools present in some of the frameworks present an alternative solution of the component application planning problem.

Deployment Runtime

The planning tools provided with this work are integrated with a deployment runtime which was designed and implemented in the work [7] and we described it in more detail in section 3.3. Support for component models can be added to the runtime via extensions of its connections and lifecycle management system. The runtime consists of multiple parts which handle various aspects of its workflow, as Target Manager, Node Manager and Execution Manager.

The deployment runtime follows the OMG D&C specification very closely and allows the use of connectors [15]. It extends the specification to allow reconfiguration of running component applications, but otherwise it uses it without significant modifications. It uses the platform independent execution data model as a common model for the execution of heterogeneous component applications. Component models supported in the runtime place additional requirements on the deployment plan being executed. This has a significant impact on our work as the deployment plan generated by our planner must fulfill these requirements. This was accomplished via component model specific extensions which modify the resulting deployment plan. The only currently supported component model of the runtime, Fractal, requires component assemblies with a running instance at runtime, which is in contrast with the component data model of the OMG specification. Our work proposes a slight modification of the component data model, which adds hybrid instantiable assemblies. This modification greatly simplified the implementation of Fractal support in the planner. We assume that the introduction of hybrid assemblies will simplify the implementation of a wide range of component models.

Deployment Factory

An alternative deployment framework is presented in [16], which aims to unify the deployment process of heterogeneous component applications. The unified deployment framework is also based on the OMG D&C specification, but it differs from it significantly. It uses a Unified Deployment Component Model (UDMC) to store component metadata. UDMC is an extension of the component data model of the OMG D&C specification so that it is compliant with it. Component model specific metadata are transformed by plugins into the UDMC. UDMC is used during planning which produces a deployment plan.

The form of the deployment plan is based on the deployment plan of the OMG D&C specification, however it is enhanced with information about component hierarchy. Thus it represents a hierarchical component application, not just a flat one. This is a significant difference from the deployment plan used by our planner and deployment runtime. This modification would simplify the deployment plans used in our deployment framework. It would remove the need for special connections or configuration properties which are used in this work to express the component hierarchy.

Connectors

Connectors presented in [15] are closely related to the deployment of heterogeneous component applications and this work. The connectors implement the communication between components leaving only business code in the components. The communication realized via connectors can implement different communication styles (procedure call, message passing, shared memory, ...) and add additional aspects to the communication (performance monitoring, logging, ...). The work [15] presents a way of defining the connectors in a template which is then used to generate the connector code. The templates are written in a domain specific language (a combination of Java and a metalanguage) which makes it easier and more user friendly to write them.

Connectors simplify deployment of heterogeneous component applications. The connector framework can automatically generate connectors which would interconnect components implemented in different component models. Our work is designed to be extensible and can be integrated with the connector framework. This integration would bring significant benefits for the deployment of heterogeneous component applications.

Performance Collection

A connector based approach to the measurement of the performance of component applications is presented in [18]. The work [18] collects and stores the required performance data via a generic measurement infrastructure. It

approaches component based applications by using measurement instrumentation based on connectors. The benefits of connector based performance collection is that they can be integrated with component applications transparently and non-intrusively.

Performance data collected with this technique could be used to enhance the planning algorithm of this work with performance aware planning heuristics. Such heuristics would allow us to implement advanced planning features, such as planning in such a way which spreads the load of component applications uniformly in the computer domain. We assume that the performance data could be used to determine the typical resource requirements of component applications. This would be used to express the resource requirements of components in a more realistic way compared to the current situation where the requirements define the worst-case scenario.

DAnCE framework

DAnCE [17] is another implementation of the OMG D&C framework with a focus on QoS (Quality of Service) and real-time systems. Currently it supports only the CORBA component model and it implements only a subset of the specification. It uses the component data model of the OMG D&C specification enhanced to describe additional concerns related to QoS and middleware configuration. DAnCE does not have an automatic planner.

Sekitei Planner

The Sekitei framework [19] solves the component planning problem with the use of techniques developed in the field of Artificial Intelligence. Its aim is the planning of component application in *dynamic component-based frameworks*. Such frameworks dynamically adapt the component applications to the changes in resource availability or client demand. The full potential of such frameworks is used when they can dynamically deploy components as a reaction to change. The Sekitei framework uses a *declarative specification* of the component application, a *trigger* which monitors the application and computer network and decides when adaptation is needed, and a *planner* which decides how to adapt the application by deploying a component.

The Sekitei planner selects components which realize the adaptation and plans them on network resources with respect to their requirements. The planner supports complex component applications and can evaluate resource requirements which are very general. However, the Sekitei planner addresses only the component planning problem, it does not handle component models or heterogeneous component applications. It does not use the OMG D&C specification for its models. It is implemented as an independent module which can be integrated with other frameworks. This presents the possibility

to integrate the Sekitei planner with the part of our work handling heterogenous component applications and their component and execution data model. Additional analysis is required to determine whether the Sekitei planner is compatible with the data models of the OMG D&C specification, especially its requirement model, and whether it could be extended to support additional technologies as connectors.

Optimal Sekitei Planner

Common planning algorithms evaluate resource requirements of components in the worst-case scenarios. The resulting plannings are non-optimal from the resource usage standpoint. An advancement of the Sekitei algorithm presented in [20] solves this by specifying the component requirements in discrete levels. The levels represent the resource usage of the component in different modes of operation (e.g. typical load vs. maximum load). The discrete requirement levels are used to find more optimal deployment plans and also to find a valid deployment plan more often in a resource constrained environment. This technique is related to the performance collection presented in [18], as that could provide it with the load levels.

Pegasus

The Pegasus project [21] focuses on computational grid environments. Component applications in that context usually perform complex data transformations like scientific simulations. The Pegasus project tries to simplify the deployment of component applications onto grids by having the users use application metadata to describe the goals of the application and inputs and output of components. A planner then uses the metadata to automatically select components which will fulfill the goals and place them in the domain respecting their resource requirements. Multiple deployment plans are found to select a high-quality solution. The goal of the Pegasus planner is more complicated than that of our planner, as it also evaluates the goals of the whole component application and constructs it automatically. Our planner uses only pre-defined component applications.

9 Conclusion

We have designed and implemented tools for planning heterogeneous component applications as specified by the OMG D&C specification. The tools provide a rich user interface which assists the user in interactively creating a valid deployment plan. The deployment plan is valid in the sense that it respects the various kinds of requirements of components.

The graphical user interface of the planning tools is integrated with an automated planning algorithm (planner) which searches for a valid planning and generates a deployment plan. The algorithm is extensible with more advanced heuristics and with support for additional technologies as connectors. Extensibility was a big focus of this work, so that the GUI can be easily enhanced to support additional parts of the deployment workflow. The GUI is based on the industry leading Eclipse platform which provides us with excellent extensibility and integration options coupled with a good user experience.

Heterogeneous component applications can be planned with the provided tools. The underlying deployment runtime places additional requirements on the deployment plan which are component model specific. The planner automatically fills the deployment plan with the required information. The mechanism for providing the component model specific metadata is extensible to support a wide range of component models and is generic enough to implement support for additional technologies as connectors. We have implemented support for the Fractal component model and tested it on several demo applications.

The platform independent data models of the OMG D&C specification can be used to store metadata of heterogeneous component applications and deploy them without the need to specialize them for concrete component models. The planner can use the specification without any significant modifications. However, we have identified a conflict between the specification's component data model and some component models as the specification supports only purely virtual component assemblies. A slight modification of the component data model which allows hybrid assemblies with their own business logic was proposed and we have demonstrated its benefits on the implementation of Fractal support.

The planning tools are a part of a deployment framework which is an ongoing research project and consists of multiple parts. Extensibility of the planner is important in respect of its further integration with the other parts. The whole deployment framework implements most of the workflow specified in the OMG D&C specification and realizes multiple additional features unforeseen in the specification and not implemented in other deployment frameworks.

10 Contents of the Distribution DVD

This work is distributed with a DVD containing the source and binaries of the implementation of the planning tools. The DVD also contains our implementation of the OMG D&C deployment framework which is necessary for the planning tools. A copy of this text in electronic form (in PDF format) and its LaTeX source is provided too. This section provides an overview of the distribution contents and of the installation and running of the bundled software. More detailed descriptions can be found in a `README.txt` file in the root directory of the DVD.

10.1 Directory Structure

The root directory of the distribution DVD contains several primary directories and a `README.txt` file containing a detailed description of the DVD:

- **deployment_framework:** This is the distribution of our implementation of the OMG D&C framework. The distribution contains the source and binaries of the various parts of the framework. Each part of the framework is stored in a separate directory containing its source and binary. The main parts of the framework must be started before the planning tools can be used. Apache Ant build scripts are used to control and build the framework.

- **gui:** The primary usage of the planning tools is via a graphical planning tool provided in the `gui` directory. The `gui` directory also contains a `README.txt` file with a detailed description of its contents. As the planner GUI is a plugin for the Eclipse platform, the DVD contains installations of the Eclipse platform for Windows and Linux, a build of the GUI plugin and its full source code. To simplify testing of the planner GUI, a bundle of the Eclipse platform (for Linux) with the planner GUI pre-installed is provided together with a sample workspace for Eclipse. The workspace contains several sample files for testing of the planner GUI.

- **requirements:** The `requirements` directory contains the required libraries (JAXB 2.1 API) and a `README.txt` file with a description of their installation. Apache Ant must be installed in your system to compile and run the deployment framework and the planning tools.

- **thesis:** The `thesis` directory contains this work in electronic form (in the PDF format) and its full LaTeX source.

- **vm:** A demo VMWare virtual machine with the planning tools is available in the `vm` directory.

10.2 Installation

Installation of the deployment framework and the planner tools places following requirements on the target system: Java SDK 1.6, Apache Ant 1.7 and JAXB 2.1 API (available from the `requirements` directory). The planner GUI is a plugin for the Eclipse platform, thus it requires an installation of Eclipse 3.3 (higher versions should work, too).

The deployment framework is installed simply by copying it (i.e. the contents of the `deployment_framework` directory) to the desired target directory. The planner GUI plugin is distributed as a JAR file stored in the `gui/planner_gui` directory. The JAR file must be copied into the `plugins` directory of your Eclipse installation. An installation of Eclipse for Windows and Linux can be found in the `gui/install` directory. The planner GUI is tested and supported on Eclipse version 3.3. However, it should be compatible with Eclipse 3.4 or higher. For easier testing, the `gui/planner_gui_bundle_linux` directory contains a bundle of Eclipse (for Linux) preinstalled with the planner GUI plugin. This bundle can be installed simply by copying it to a directory.

10.3 Running the Planning Tools

The deployment framework must be running before the planning tools can be used. You must start the component repository, target manager, node managers and the execution manager for the full functionality. All these parts of the framework are started via Apache Ant scripts from their respective subdirectories in the `deployment_framework` directory.

The planning GUI is started simply by running Eclipse with the planning GUI plugin installed (as described in section 10.2) and selecting the `Deployment` perspective. If the perspective is not selected by default (i.e. in the Eclipse and planner GUI bundle), you can select it via the Window-Open Perspective-Other-Deployment menu of Eclipse.

A sample Eclipse workspace for use with the planner GUI is provided in the `gui/workspace` directory. The whole workspace should be copied to a desired directory and selected during the startup of Eclipse. The workspace contains a `README.txt` file with a more detailed description and multiple example files related to planning (deployments and deployment plans).

A demo virtual machine is provided on the distribution DVD in the `vm` directory. The virtual machine contains a Linux system with the deployment framework and the planner GUI pre-installed. After startup of the virtualized Linux system, the user is presented with a desktop environment offering icons to run all the parts of the deployment framework and to run the GUI. The virtual machine is runnable via products of VMWare [31], i.e. the free VMWare Player[32] or VMWare Server[33].

References

[1] EJB, http://java.sun.com/products/ejb/

[2] CORBA, http://www.omg.org/technology/documents/formal/components.htm

[3] Fractal, http://fractal.objectweb.org/specification/index.html

[4] SOFA, http://dsrg.mff.cuni.cz/projects/sofa/tools/doc/compmodel.html

[5] SOFA2, http://sofa.objectweb.org/

[6] Object Management Group: Deployment and Configuration of Component-based Distributed Applications Specification, OMG document ptc/05-01-07, January 2005

[7] Šafrata, P.: Infrastructure for Deployment of Heterogeneous Component-based Applications, Charles University, Prague, Sep 2007

[8] Roman Barták, lectures: http://ktiml.ms.mff.cuni.cz/ bartak/planovani/prednaska.html

[9] M.Ghallab, D.Nau, P.Traverso: Automated Planning: Theory and Practice, http://www.laas.fr/planning/

[10] Joseph Y-T. Leung: Handbook of Scheduling: Algorithms, Models, and Performance Analysis, http://web.njit.edu/ leung/handbook

[11] Philippe Baptiste, Claude Le Pape, Wim Nuijten: Constraint-Based Scheduling - Applying Constraint Programming to Scheduling Problems, 2003

[12] Eric Clayberg, Dan Rubel: Eclipse - Building Commercial-Quality Plug-Ins, 2006

[13] Jeff McAffer, Jean-Michel Lemieux: Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications, 2005

[14] OSGi, www.osgi.org

[15] Malohlava, M.: Using Stratego/XT for Generation of Software Connectors, Charles University, Prague, Jan 2007

[16] Hnetynka, P.: Making deployment process of distributed component-based software unified, Charles University, Prague, Sep 2005

[17] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale, DAnCE: A QoS-enabled Component Deployment and Configuration Engine

[18] Bulej, L.: Connector-based Performance Data Collection for Component Applications, Charles University, Prague, Jul 2007

[19] Tatiana Kichkaylo, Vijay Karamcheti: Optimal Resource-Aware Deployment Planning for Component-Based Distributed Applications

[20] T, Kichkaylo, A. Ivan, and V. Karamcheti: Constrained Component Deployment in Wide-Area Networks using AI Planning Techniques

[21] Blythe, J., et al. 2003. The role of planning in grid computing. Proc. ICAPS. http://citeseer.ist.psu.edu/blythe03role.htm

[22] Model Driven Architecture, http://www.omg.org/mda/

[23] Java Architecture for XML Binding, https://jaxb.dev.java.net/

[24] Apache Ant, http://ant.apache.org/

[25] log4j, http://logging.apache.org/

[26] Standard Widget Toolkit, http://www.eclipse.org/swt/

[27] MetaObject Facility, http://www.omg.org/mof/

[28] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/

[29] Frank Budinsky, Dave Steinberg, Ed Merks, Ray Ellersick, Timothy J. Grose: Eclipse Modeling Framework, 2003

[30] Graphical Modeling Framework, http://www.eclipse.org/modeling/gmf/

[31] VMWare, http://www.vmware.com/

[32] VMWare Player, http://www.vmware.com/products/player/

[33] VMWare Server, http://www.vmware.com/products/server/